Developing an In-kernel File Sharing Server Solution Based on Server Message Block Protocol

Bastian Arjun Shajit

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 25.09.2016

Thesis supervisors:

Prof. Raimo A. Kantola

Thesis advisor:

M.Sc. Szabolcs Szakacsits

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Bastian Arjun Shajit

Title: Developing an In-kernel File Sharing Server Solution Based on Server Message Block Protocol

Date: 25.09.2016 Language: English Number of pages: 8+79

Department of Communications and Networking

Professorship: Networking Technology

Supervisor: Prof. Raimo A. Kantola

Advisor: M.Sc. Szabolcs Szakacsits

Multi-device and multi-service smart environments make heavy use of the Internet and intra-net, thus constantly transferring and saving large amounts of digital data leading to an exponential data growth. This has led to the development of network storage systems such as Storage Area Networks and Network Attached Storage. Network Attached Storage provides a file system level access to data from storage elements that are connected to the network. One of the most widely used protocols in network storage systems, is the Server Message Block(SMB) protocol, that interconnects users from various operating systems such as Windows, Linux and Mac OS. Samba is a popular open-source user-space server, that implements the SMB protocol. There have been a multitude of discussions about moving traditional user-space applications like web servers to the kernel-space in order to improve various aspects of the server like CPU utilization, memory utilization, memory footprint, context switching, etc.

In this thesis, we have designed and implemented a server in the Linux kernel space. We discuss in detail, the features and functionalities of the newly implemented server. We provide an insight into why some of the design considerations were made, in order to improve the efficiency of protocol handling by the in-kernel file sharing server. We compare the performance of the user-space Samba solution with the in-kernel file sharing solution, implemented and discussed in this thesis, against different workloads to identify the competitiveness of the developed solution. We conclude by discussing what we learned, during the implementation process, along with some ideas for further improving the feature set and performance of the in-kernel server solution.

Keywords: Network storage systems, Server Message Block, Linux-kernel, user-space, Samba, server, embedded, enterprise

Foreword

This thesis is written in partial fulfillment of the requirements for a master's degree in *Communication Engineering*, specializing in *Networking Technology* at Aalto University. Primary research and implementation of the network protocol discussed in this thesis were done at Tuxera Inc, and all other activities were carried out under the supervision of Prof. Raimo A. Kantola, between March 2016 and September 2016.

Like any other project, my thesis wouldn't have taken shape without the constant support from various individuals. I would like to extend my sincere thanks to all of them without being my chatty self.

First, I would like to express my deepest appreciation to Prof. Raimo A. Kantola, for giving me the opportunity to work under his supervision. He provided invaluable suggestions, guidance and was always patient during the various phases of review of this literature.

I would like to thank some of my colleagues from Tuxera Inc. Szabolcs Szakacsits, the Founder, President and CTO, whose open mindedness and continuous support provided me with the opportunity to publish a part of my work; Oleg Kravtsov, our team lead and other team members specially Boris and Maksim, for without them we never would have had a working prototype on time.

I sincerely thank my friends, in Finland, whose constant nagging, guilt tripping and negative reinforcement, worked surprising wonders in pushing me to complete this thesis.

And most importantly, I would like to thank my parents, for their support.

Espoo, 25.09.2016

Bastian Arjun Shajit

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

AFP	Apple Filling Protocol
API	Application Program Interface
CIFS	Common Internet File System
DAS	Direct Attached Storage
DUT	Device Under Test
DCE/RPC	Distributed Computing Environment / Remote Procedure Calls
eSATA	External Serial Advanced Technology Attachment
FC	Fibre Channel
FCIP	Fibre Channel over IP
FQDN	Fully Qualified Domain Name
GbE	Gigabit Ethernet
GSS-API	Generic Security Service Application Programming Interface
HBA	Host Bus Adapter
iFCP	Internet Fibre Channel Protocol
IO	Input Output
IP	Internet Protocol
iSCSI	Internet SCSI
JBOD	Just a Bunch Of Disks
NAS	Network Attached Storage
NetBEUI	NetBIOS Extended User Interface
NetBIOS	Network Basic Input/Output System
NFS	Network File System
NTLM	NT Lan Manager
NTLMSSP	NT LM Security Support Provider
OpenSSL	Open source Secure Sockets Layer
RAID	Redundant Array of Inexpensive Disks
RDMA	Remote Direct Memory Access
RoCE	RDMA over Converged Ethernet
SAN	Storage Attached Network
SAS	Serial Attached SCSI
SATA	Serial Advanced Technology Attachment
SCSI	Small Computer System Interface
SMB	Server Message Block
SPNEGO	Simple and Protected GSS-API Negotiation
SSD	Solid State Drive
TCP	Transmission Control Protocol
UNC	Universal Naming Convention
VFS	Virtual File System

1 Introduction

With the advent of the Internet of Things, Big Data and a rapid development in the field of embedded electronics and software technologies, there has been an increase in the use of smart, low-end and high-end devices, both at home and at enterprises in the modern environment. A study[1], has revealed that the digital data is always expanding, and is expected to grow by a factor of 300, between 2005 and 2020. The study predicted that more than 68% of the data being generated, is mostly due to digital media streaming, and image transfers. It also claims that enterprises are responsible for almost 80% of this data, which traverses their networks and server farms during end-user activities. The study also predicted that the amount of data generated might even surpass the available storage capacity in less than a year. This boom in data and ever increasing amounts of digital data places very high requirements and expectations on the storage systems.

1.1 Motivation and aims

Storage systems have evolved over the past few decades and have become fundamental in a digitized framework. General end users, small scale businesses and enterprises are over-whelmed by this explosive storage growth. With improper technical expertise, it becomes extremely daunting to manage such rapid growth in data. Many of the digitized frameworks save large amounts of data in some form of storage systems. These storage systems are usually local storage media such as hard disks, USB flash disk, solid state drives, CDs, etc. With an ever increasing demand for storage, this approach is impractical because it is inefficient and costly, makes it difficult to manage and protect large amounts of data when they are scattered all over, and carrying around arrays of disk to get access to their data is inconvenient for the general consumers. Due to these reasons, network storage systems have gained popularity over the past few years.

A network storage system may be decentralized or distributed. In a distributed storage system, an entity can get access to stored data over the network. Due to the many advantages a distributed storage system provides, such as availability, scalability and reliability, network storage systems have gained widespread popularity among consumers and enterprises. With the advent of high speed local area networks and technologies such as 1 GbE and 10 GbE becoming commodities, network storage systems have become increasingly deployable in many networked environments. Usage of file sharing protocol is one among the plethora of technologies that make network storage systems possible. Such network storage systems provide file system capabilities over the network through which the users are able to access files and its respective meta-data such as size of the file, last access time of the file, etc., from the server using the underlying file sharing protocol.

One of the primary expectations from these file sharing protocols, is that the network storage systems should be able to store and access information with very low latency and response time. The file sharing protocols are continually improved with new features based on popular demands from general consumers, network hardware & storage hardware vendors and finally to accommodate advances in network architectures. Over the hue and cry for better hardware and efficient storage systems, it is very easy to overlook the performance of the software that implements the file sharing protocol. The software implementing a file sharing server must be designed and optimized for performance considering the system behaviour and architecture.

Server Message Block protocol is one such file sharing protocol that can be found in many network storage systems, with each operating system supporting their own custom implementation of the protocol. Different operating systems such as Windows, Linux and Mac OS, either have their own implementations of the file sharing protocol in user-space or directly integrated with the operating system. Recent study[1] by the Linux Foundation has shown that the deployment of Linux as an enterprise grade operating system has been increasing, and more than 75% of the servers hosting cloud platforms are based on Linux. Additionally, many readily available consumer grade file sharing servers are also based on Linux. Despite the increase in popularity, Samba[2] is the only freely available open source project that provides a fully functional Server Message Block based file sharing server in the user-space.

With Linux gaining popularity both in consumer and enterprise markets, we believe that there is a potential for providing a solution that can perform better than the existing solution in the user-space. In this thesis, we propose a file sharing solution which resides in the operating system or the Linux kernel. The proposed

[1] https://www.linux.com/publications/2014-enterprise-end-user-report, The user trends are bound to change and as such these documents may not be available or deemed invalid at later point of time

[2] https://www.samba.org/

solution has been developed after carefully researching the features and capabilities of the existing user-space solution. As a part of the thesis, we compare the existing user-space solution against the newly developed in-kernel file sharing solution. The competitiveness of the newly developed solution is demonstrated by analysing the system architecture, features, functionalities, test methodologies and performance against different workloads.

1.2 Author's contribution

This thesis work has been carried out in Tuxera Oy, Finland under the supervision of Szabolcs Szakacsits. The author along with a team of 3 other core members, was responsible for designing, implementing, analysing & fixing bugs, performance benchmarking and testing the in-kernel file sharing server. The author was also primarily responsible for analysing and improving performance of the kernel file sharing server.

Due to company policies and reasons pertaining to responsible disclosure, some of the information related to analysis strategy, system & network traces, actual behaviours, details of testing framework, a detailed explanation of different components involved in the implementation and codes corresponding to the in-kernel file sharing solution cannot be revealed in a public document, e.g. this thesis. We believe that the information included in this thesis should be sufficient to provide a detailed picture of the architecture, and to assure the readers of the advantages of moving the file sharing server to the kernel space.

1.3 Structure of thesis

The remainder of the thesis is divided into 5 chapters.

Chapter 2 provides the essential background regarding the different storage systems and their main characteristics.

Chapter 3 explains the key characteristics of the Server Message Block protocol. It provides an overview of general communication scenarios between the client and server, and briefly describes the existing implementations currently available at the time of writing this thesis. This chapter provides references to many open protocol specification documents that were made available by Microsoft. At the time of writing this thesis, the revision of the specification documents is v20160714. Since Microsoft is continuously updating this document, the revision number is bound to be continually updated and the version used may not be available to the public at a later point in time.

Chapter 4 proposes the file sharing solution developed for the Linux kernel. It provides a detailed description of each of the components involved in the architecture, and provides an overview of the features and functionalities of the newly proposed solution.

Chapter 5 evaluates the performance of the in-kernel file sharing server, and compares it with the existing user-space solution. It describes in detail, the environment and the design of the benchmarking suite along with the reason for choosing the server and client hardware used for performance evaluation.

Chapter 6 discusses the results seen during performance evaluation, and attempts to reason about the observed results from the previous chapter. It briefly discusses the new development activities that are being or have already been undertaken, to further improve the performance of the in-kernel file sharing solution.

2 Background

With rapid development in technologies that aid the growth of Internet and network technologies, such as e-commerce, data warehousing, and content delivery networks, there has been a rising demand for enterprise grade storage capacity, by leaps and bounds. Information has become an essential part of day to day activities, and hence it is of utmost importance to save the information gathered from different sources, and also provide access to them in a fast and reliable manner. Hence data transmission, data storage and data processing have become the pillar stones for the existence of most technologies, thereby becoming the primary yardstick on which newer solutions are developed.

For the design, realization and deployment of modern storage systems, it is therefore important to study and understand the basic properties of different storage architectures and the various components used to implement them. This chapter serves as a primer to the modern-day storage technologies.

2.1 Introduction to data storage

Storage systems can be classified based on two factors[3]: (1) Physical connectivity and the type of protocol used by the connecting media, (2) The type of IO requests that are communicated over the media, which is either block-level IO request or simple file IO request. Based on the above factors, modern day storage systems can be broadly classified into two categories, namely *Direct Attached Storage (DAS)* and *Network Storage Systems*. The Network Storage Systems can be further classified into *Storage Area Networks (SAN)* and *Network Attached Storage (NAS)*.

[3] http://bnrg.cs.berkeley.edu/~randy/Courses/CS294.S13/12.1.pdf

2.1.1 Direct Attached Storage

As the name suggests, DAS is the traditional computer storage system, in which a storage media is directly attached to a host such as a PC or a server. This storage device, may comprise of one or more storage drives connected via a Host Bus Adapter (HBA). The HBA provides interfaces such as SATA, eSATA, SCSI, Serial Attached SCSI (SAS), Fibre channel or some other variations of external disk enclosures containing multiple storage drives in the form of just-a-bunch-of-disks (JBOD) or RAID configurations.[4]. The architecture is shown in Figure 1.

DAS has many advantages, such as the ease and lower expense for setup and installation, relatively faster access to on-disk data since the device is directly attached to the host and easy technical know-how. It is also marred by some serious drawbacks, the first being scalability and expansion. Secondly, resource sharing is not possible with multiple hosts since they are dedicated and therefore, unused capacity is lost when other hosts need them. Thirdly, due to the lack of a central management system and network infrastructure, server resource information is kept locally and is not available to other hosts[4].

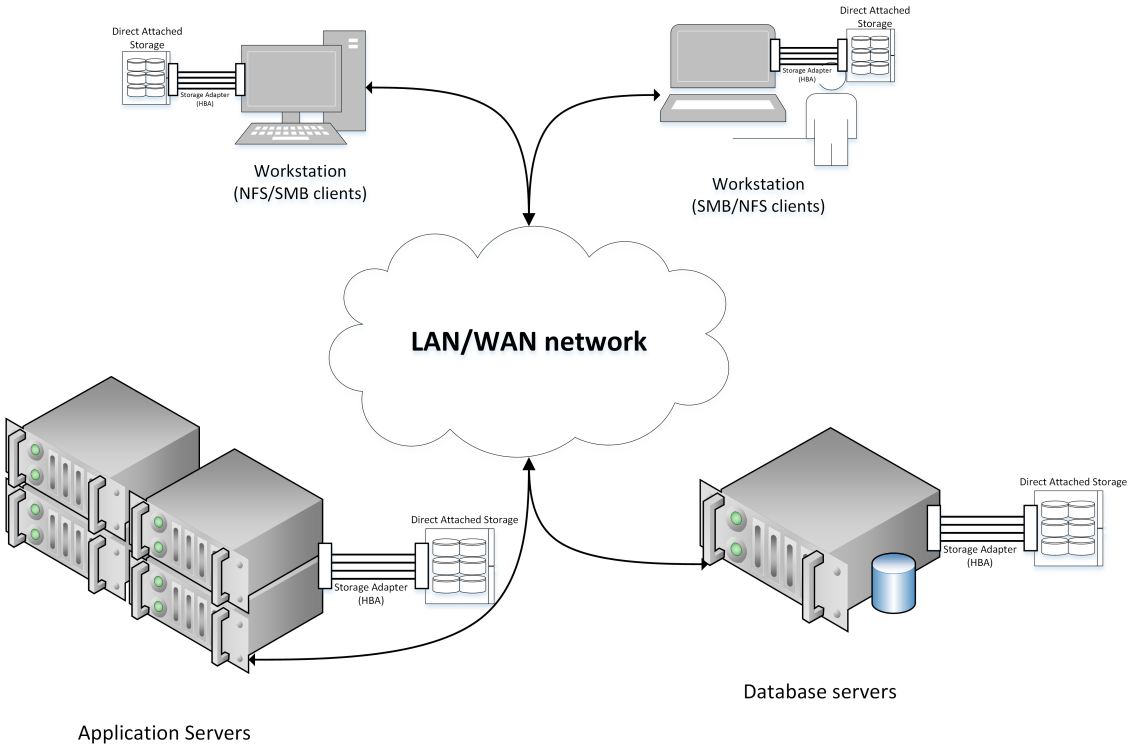


Figure 1: Typical DAS architecture

In terms of application, DAS has mainly been chosen because of its speed and lower

[4] https://technet.microsoft.com/en-us/library/dn610883(v=ws.11).aspx

latency in comparison to network based storage systems. DAS systems are mainly deployed in small scale businesses where file sharing is mostly local and having very few clients and servers involved in the information sharing[5].

2.1.2 Network Storage Systems

Network storage systems are classified based on the type of IO protocol i.e. the type of IO request that is used over the media, where media refers to the type of connection between the host and the storage device, and the protocol used in the connection. From the user/client's perspective, the primary difference between different network storage systems, is based on the way the storage device is abstracted over the network.

They can be broadly classified into: *(1) Storage Area Networks(SAN)* and *(2) Network Attached Storage(NAS)*. SAN exports the storage media as a block addressable device, which is a lower level of abstraction similar to seeing a disk. NAS provides a much higher level of abstraction known as a file-system[2].

Storage Area Networks

SAN is a dedicated, extremely fast private network that connects the storage devices. It may be an individual storage device, a disk enclosure, a disk array attached to a multi-purpose server via an optical transport fabric (Fibre channel interface) through which it is able to assimilate distributed storage appliances. SAN provides a complicated architecture through which host computers treat the remotely attached storage device as a DAS. This allows the host to perform block level access on the storage device. Using the Fibre channel technology, the hosts and clients get fast and reliable communication and data transfer that facilitates simultaneous asynchronous access among multiple hosts. The architecture is shown in Figure 2.

Some of the advantages of SAN compared to the traditional DAS architecture are that, it introduces the concept of network management in the storage paradigm thus facilitating high speed data sharing, data mirroring, migration and fail-safe handling[5]. Since SAN allows Fibre Channel interface between the storage devices and the servers, the two entities can be present in entirely different locations. The number of storage arrays on a specific host can be reduced, since different storage appliances can be connected to share their capacities. The main disadvantage for SAN deployment is that, it requires specialized hardware for handling Fibre Channel

[5] http://bnrg.cs.berkeley.edu/~randy/Courses/CS294.S13/12.1.pdf

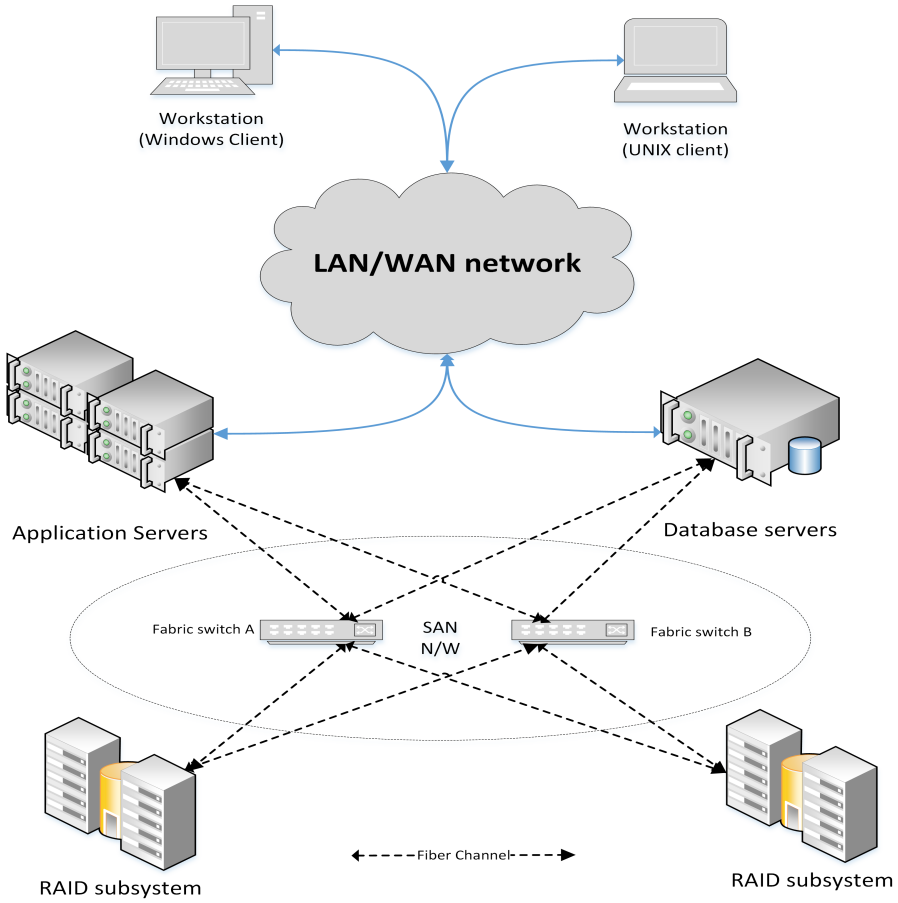


Figure 2: Typical SAN architecture

based networks. As a result, the initial cost of deployment is extremely high and requires special technical know how to operate the infrastructure[6].

Over the years, SAN has developed from the more traditional FC-SAN configuration requiring special hardware such as HBA and FC switches, to the lower-cost alternative known as IP SAN which is less complex and easier to manage. Some of the main protocols used in IP SAN are iSCSI, FCIP (Fibre channel over IP) and iFCP (Internet Fibre channel Protocol), which encapsulate protocol specific block IO commands over TCP/IP and use standard Ethernet for data transmission. Since IP SAN uses commodity networking hardware for transmission, it introduces higher latency compared to FC SAN.

[6] http://bnrg.cs.berkeley.edu/~randy/Courses/CS294.S13/12.1.pdf

Network Attached Storage

NAS devices can be considered as network appliances with one or more storage devices, thereby providing access to data for many heterogeneous clients. Compared to the block level IO requests that are supported by DAS and SANs, NAS devices handle only file IO request. File IO appears at a higher level of abstraction compared to SAN, and accesses files/directories rather than raw storage. Therefore, unlike DAS or SAN, the storage shared by NAS devices on the network has no awareness of the existence of logical disk volumes. NAS systems can be considered to be dedicated file servers running a custom operating system. These file servers support disk storage for exporting storage over the network, and a plethora of file systems internally, to control the way data is stored and retrieved from the locally attached storage. A typical NAS architecture is shown in Figure 3.

NAS devices in the network reduce the load on the servers in the network by taking over responsibilities of file sharing. NAS systems are usually preferred because they can act as general purpose file-servers, and are easy to configure and administer. Since NAS exports file level access, it allows setting disk quotas, user permissions & ACLs, folder privileges, etc. NAS systems have many advantages such as the ease-of-installation which allows it the privilege of plug and play, low maintenance and high scalability. NAS, if configured properly, allows resource consolidation and pooling, thus allowing different users accessing the same file system over the network to be allocated a space within the same volume on demand, rather than allocating a dedicated volume to the user as in DAS and SAN. This allows multiple users to share the same storage device. Since NAS storage systems can easily plug into any network, it may not enjoy the privilege of a dedicated network for IO, like in SAN. As a result, if there are multiple users on the same network, the performance of NAS could be limited. Unlike SAN, which has a dedicated hardware for protocol handling, NAS relies on the host's network stack which could introduce a significant overhead. In order to reduce this overhead, a popular option has been to offload the processing of TCP/IP stack from the host to its network controller. Due to its simple nature of providing file level access, reduced complexity, scalability and ease of management, NAS has been an ideal choice for scaling storage capacities in many data centres and organizations.

NAS uses file based protocols such as SMB/CIFS (popular among different versions of Windows operating system), AFP (Mac OS) and NFS (Linux specific) for handling management of IO requests to storage media connected over the network. These are protocols implemented by default in many operating systems and hence are readily available for everyone. This flexibility allows setting up NAS devices on the fly, thus facilitating even a low-end, general purpose PC with only a single internal storage disk to act as a NAS server.

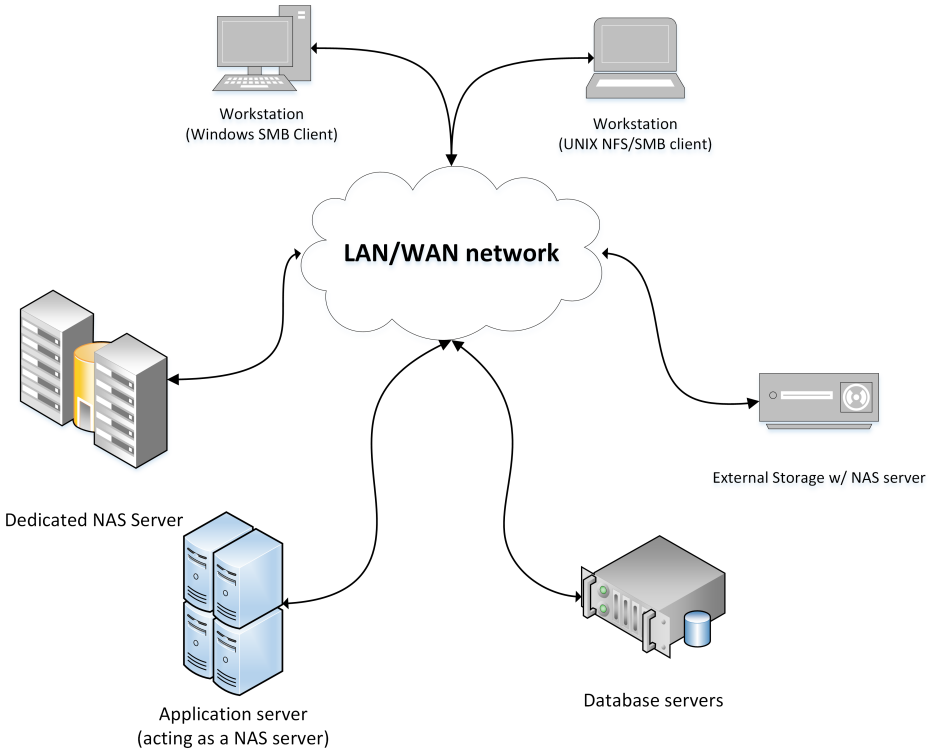


Figure 3: Typical NAS architecture

2.2 File System versus NAS

Before we proceed with the introduction of one of the widely used NAS protocols, it is important to understand some of the basic differences between the file-system abstraction that is provided by NAS protocol and the local file system.

Figure 4 represents an abstraction that is exported by the local file system to an application running on a workstation. We can interpret the following from the local file system abstraction. Firstly, any local application can invoke a normal file system operation like creating, deleting/removing, renaming files or directories, reading from files, writing to files, etc. Secondly, on accepting an IO request, the IO manager or VFS (Virtual File System) then invokes file system specific operations. Thirdly, the file system converts these requests into block IO requests, which are then translated by hardware specific device drivers into device specific commands. Finally, the request is processed and the result is returned to the application that invoked the request in the first place.

Figure 4 is a very simple representation of how a local storage file system would look like to an application running on the same host. The way a network storage system abstracts file system functionalities to a remote client is shown in Figure 5. This diagram is only a reference and is a very close representation of a network

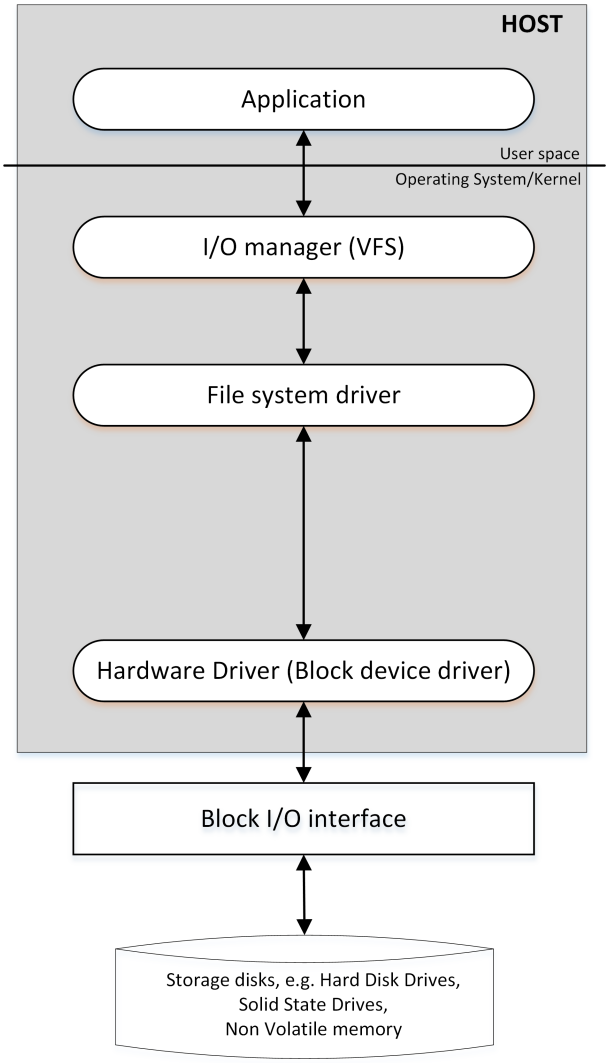


Figure 4: Conceptual Model of a local file storage abstraction

storage system's architecture. This figure also represents possibilities where there may be simultaneous accesses from multiple applications running in many different workstations.

We can deduce the following from the conceptual network file storage abstraction model. All applications, either directly or indirectly, can invoke different file or directory operations. The VFS or the IO manager understands that the access to a file or directory is residing on a remote volume and invokes the corresponding network file system for further operations. The IO operations are only possible if the remote volume is already mounted locally and is seen as a local file system. The concept of mounting the remote file system will be clarified in the later sections.

On receiving an IO request from the local application, the network file system driver residing on the client host translates the request into corresponding network protocol

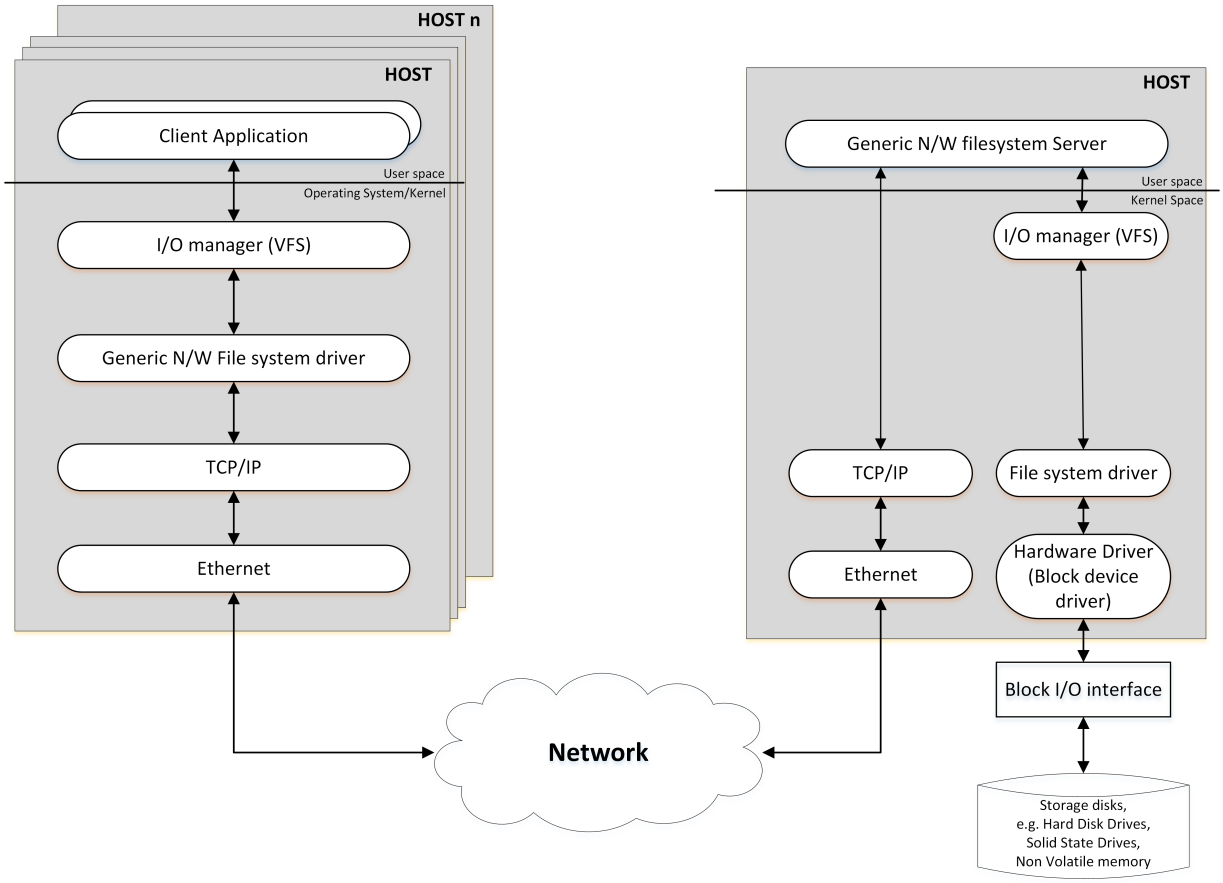


Figure 5: Conceptual Model of a Network file storage abstraction

data units, and forwards the request to the server. The server interprets this request as a local file system operation, and forwards the translated network request to the IO manager or VFS in the server. On accepting the IO request, the IO manager or VFS (Virtual File System) then invokes file system specific operations. The file system converts this request into block IO requests, which are then translated by hardware specific device drivers into device specific commands used to access data from disks connected to the server. The request is processed and the result is returned to the requesting application, which in this case is the server application. The server application in turn, responds to the remote client's request.

Though the overall aspects of the the two models shown in Figure 4 and Figure 5 might look the same, there are some very important differences between the two. Firstly, in case of the local storage model, where the host operating system takes care of scheduling, it can be easily configured so that certain patterns of IO requests or some desired application can be given a priority, but in a networked model there are many applications running concurrently in different clients and may not have any knowledge about each other. Secondly, in a networked model, the number of layers of virtualization and abstraction across which a single request might traverse,

is not predictable as opposed to the predictable nature of the exact number of layers of abstraction that a local IO request passes through. The above Figure 5 is used superficially to understand the underlying principles in a simple way, but the real-world implementations are much more complex.

2.2.1 NAS File System Characteristics

We know by now that when dealing with network file systems, there may be many layers of abstraction that are transparent to the application. Most network file systems reside in the presentation and application layers of the network models, and mostly rely on the underlying transport layer and network layer for reliable delivery. There are some key characteristics that are useful in identifying network file systems[6]:

Firstly, **File access vs Block access**; As mentioned earlier, network file systems provide access to files and as such provide a notion of familiarity while dealing with files when compared to the network block device protocols such as SAN, which provides access to raw blocks of the storage device.

Secondly, **Abstraction**; Network file systems provide transparency such that local applications can work seamlessly as if they were working with a local file or directory.

Thirdly, **Functional Synonymity**; NAS file systems often have similar operational parity as that of a local file system. As a result, any local file system operations may have a one to one proximity to the requests performing the same operation over the wire on a remote file system.

Fourthly, **Optimization**; An application's workload requests often closely match the requests that are sent to the server as opposed to network block protocols. This allows for interesting optimization techniques based on the knowledge of local file system access patterns.

Lastly, **Convergence**; There is a pot-pourri of operating systems out there with each operating system having its own arsenal of file systems. NAS file system provides an abstraction, wherein the client workstation can have consistent file access without needing to care about the remote server's operating system or the storage media.

2.3 Introduction to SMB

With a growing need for storage, users require more storage than their hardware can accommodate. For this reason, many remote servers use protocols that masquerade their storage as a local file system on any workstation that requires more storage space. One of the most prevalent protocols used by the servers for this purpose is the CIFS/SMB protocol.

SMB, an acronym for Server Message Block, is a state-ful client-server protocol that tries to provide most of the functionalities offered by a local file system. It is a protocol used for sharing files and directories, exporting printers and serial ports and for providing a communication abstraction for named pipes and mail slots remotely accessible across different computers. SMB, by no means is the only network file system protocol, and there are rich alternatives available based on the operating system being used. While referring to this protocol, the term CIFS (Common Internet File System) has become obsolete and only the term SMB will be used from here on to refer to this protocol, unless we are discussing about some version or dialect specific feature of the protocol. CIFS is not only a term to denote SMB protocol but is also a dialect of SMB.

The following sections provide a brief overview on the history and protocol family relevant to the background of SMB and this thesis.

2.3.1 History

The core SMB protocol was invented by Dr. Barry Feigenbaum from IBM laboratory[7]. It was initially named as BAF protocol and renamed to SMB before the official release. The earliest documents on SMB was published by IBM as a part of IBM Personal Computer Seminar proceedings in 1984. Following this, a much detailed LAN technical reference for the SMB transport, namely NetBIOS was published a few years later[6]. NetBIOS is a session layer protocol that has been implemented over IPX/SPX and DECNet. NetBEUI was later released in 1985 as an extended version of NetBIOS by IBM to run over its enhanced token ring network. To top the plethora of confusing naming conventions, Microsoft named its own implementation of NetBIOS Frame protocol, also known as NBF, to run over IEEE 802.2 LLC as NetBEUI. Due to the limited routing capabilities of the NBF/NetBEUI protocol, later versions of SMB used an alternative transport mechanism by encapsulating NetBIOS over TCP/IP, documented as part of RFC1001 and RFC1002 in 1987.

The protocol was then developed further by Microsoft with support from Intel and 3Com. They were responsible for creating newer dialects of the protocol and for

[7] https://www.samba.org/samba/docs/myths_about_samba.html

releasing technical documents periodically. Microsoft was responsible for adding large and significant extensions to the protocol along with some substantial contributions from other parties like IBM.

Microsoft made LANMAN1.0, a dialect of SMB, its first default dialect on OS/2 (operating system by Microsoft and IBM). The protocol at its prime, lacked any sort of authentication support and provided very little security. By the early 1990s, SMB became the default protocol for DOS and Windows operating systems. SMB protocol started finding applications not only as a file sharing protocol, but also for network resource discovery, network browsing, networked printers, remote management of servers, etc. In order to support these additional functionalities, SMB protocol was further extended to support various network IPCs such as Mailslots, RPCs and named pipes. This led to the development of more sophisticated IPCs, that allowed DCE/RPCs[8] to use SMB Named pipes as transport.

Around 2002, SNIA released its first CIFS technical reference[7] which described in detail almost all of the extensions introduced by Microsoft for Unix and Mac OS. Since most of the operations done by SMB protocol are not POSIX[9] compliant, and as a result CIFS extensions were introduced to the CIFS dialect of SMB. This allowed any operation on the server to behave similar to a POSIX API.

Some of the well known NAS file systems apart from SMB are: AFP, also known as the Apple Filing Protocol supported by Apple in its Mac OS and Network File System (NFS) supported by all flavours of Linux. Both Mac OS and Linux have been steadily gaining popularity in both the consumer and enterprise environment as more and more users are joining the networked ecosystem. But the most popularity for SMB came when Apple ditched further development of AFP and made SMB its default file transfer protocol around the year 2013 in Mac OS X Maverick.

SMB is under constant development, and over the years many new versions of the protocol have been introduced along newer dialects, which brings performance enhancements and other features that are required for scalability, reliability and better security (authentication and encryption). At the time of writing this thesis, SMB2 protocol version is under heavy development, with SMB 3.1.1 dialect introducing some very significant clustering and security enhancements. Since the CIFS UNIX extension is only applicable for SMB1 dialect, there are discussions of extensions to SMB2 protocol version in order to make it POSIX compliant. As the technical aspects of the extensions become clearer, improved and official technical documents are being considered for release.

[8]Primary speecification of DCE 1.1, http://pubs.opengroup.org/onlinepubs/9629399/

[9]In this thesis, we use "POSIX" to refer to the API semantics that a POSIX-compliant operating system implements as per the open group base specifications.

2.3.2 Version and Protocol family

SMB protocol is being continually improved and hence newer versions and their corresponding dialects are being introduced, as more functionalities are added to portray significant improvement in the protocol architecture. From Table 1, we can infer that a version represents a major overhaul of the protocol and each dialect revision adds some significant features to the protocol. For the sake of discussion, Table 1 does not show older and deprecated variants and presents only the most relevant and recent versions and variants.

Table 1: Protocol versions and corresponding dialects

Version	**Year**	**Dialect revision**	**Comments**
SMB	1995	CIFS	NT lan manager extended protocol
	2000	SMB 1.0	Extension to the CIFS protocol to add additional transport, authentication methods and capability negotiation
SMB2	2006	SMB 2.0.2	New version with major redesign of SMB, improved scalability, minor performance enhancements and better encryption support.
	2009	SMB 2.1	File leasing, Large MTU support and peer content caching
	2012	SMB 3.0	Many enterprise file sharing features for high availability, performance enhancements such as multichannel RDMA transport, scale-out and better security in terms of encryption and signing
	2013	SMB 3.0.2	Improvements over SMB 3.0
	2015	SMB 3.1.1	Includes pre-authentication integrity check, encryption and cluster improvements

3 Server Message Block protocol

Regardless of the version or dialect, the primary function of the SMB protocol is to provide file sharing capabilities. In addition, the SMB protocol includes many complimentary functionalities, namely network browsing, printing over a network, authentication, file and record locking, change notifications and opportunistic locks.

Network Browsing

This capability allows any host to determine the presence of SMB servers and shared resources in the same network or across subnets.

Dialect Negotiation

Different clients and servers may choose to implement different versions and dialects of the SMB protocol. Dialect negotiation allows any clients to select a particular version of the SMB protocol supported by the server. This ensures that the two parties involved in the communication, namely the client and the server, are aware of the features and functionalities each of them supports. This capability allows for backward compatibility between new servers & older clients and vice versa.

Network Printers

Network Printers work by using *spoolss* named-pipe endpoint, which lets the client communicate with the Print Spooler services supported by the server via RPC. The capability allows access to shared printers available in the network, so that the clients can then use them like a local printer resource.

Authentication

The protocol provides various levels of authentication at the user level. These levels help maintain access control lists for the users who can authenticate and use the resources exported by the server. This also enables the server to have file/directory level access control entries.

File and record locking

The SMB protocol provides two type of locking capabilities for the client. First, *record locking* which allows the clients to lock a specific range of bytes within a file. Second, *deny modes* that are most often specified at the time of opening the file. This allows client applications to limit the type of simultaneous accesses on the file.

Change notification

Change notification is the capability that automatically informs the clients about changes to the metadata corresponding to events, such as renaming a file, deletion of a file, etc, associated with the files/directories residing in the server.

Opportunistic locks

Opportunistic locks are meant for the purpose of enhancing the performance of clients over the network, by providing different means to cache the file locally at the client side. Some of the opportunistic locking capabilities supported by the protocol are lock caching, write caching and read-ahead.

The capabilities listed above are common to all the different dialects and versions of the SMB protocol, as introduced in the earlier sections. This chapter acts as a technical guide to the otherwise extremely confusing nature of the technical specification of the corresponding protocols. Henceforth, any reference to SMB2 will only refer to the technical document defining SMB2[10], and any reference to SMB1 will refer to the technical documents defining CIFS[8] and its extensions defined by the SMB1 document[9]. SMB protocol will hereafter be used to collectively refer to all of the protocol as defined in [8], [9] and [10], unless explicitly stated as part of a particular standard. This chapter provides an in-depth overview of other protocols and services that the SMB protocol relies on, and the corresponding set of packet exchanges that enable seamless functioning of these services.

3.1 Protocol dependency

In the TCP/IP model, the SMB protocol is often found in the Application layer and in the OSI model, it resides in the Application and Presentation layer. The SMB protocol, hence relies on the lower layers of the network model to take care of the transport.

SMB protocol usually uses Direct TCP as its transport layer protocol, as shown in Figure 6. It also uses NetBIOS over TCP primarily for the purpose of backward compatibility, since all the latest versions, namely SMB2[10] and SMB1[9] can be used directly over TCP. SMB2 is not limited to TCP and NetBIOS as its primary transport, but can also include Remote Direct Memory Access[12]. RDMA as a transport is

implemented over Ethernet, also known as RDMA over converged ethernet[13, 14], or over Infiniband network[15], or on iWARP, which defines RDMA over connection oriented transport like TCP[16, 17].

SMB protocol primarily relies on two different services or resources for seamless functioning. One of them is a reliable transport for delivery of messages in the correct order. Three of the available transports are Direct TCP, NetBIOS over TCP and RDMA. In this thesis, both RDMA and NetBIOS are out of scope since most of the available clients that use operating systems such as Windows, Linux or Mac OS use Direct TCP as their preferred transport and hence the implementation in this thesis is also limited to Direct TCP. The other service it relies on, is a robust security infrastructure to support authentication mechanism using Simple and Protected GSS-API Negotiation (SPNEGO), as defined in RFC4178 and in its technical specification[18].

3.1.1 Transport: NetBIOS over TCP/IP

Though its not the default transport in many available SMB implementations, NetBIOS still has a lot of relevance in the general functioning of the SMB protocol. The original NetBIOS specification defines methods by which an application can employ NetBIOS as a software interface and provides an outline for naming convention of services. In other words, it only defines the interface and services available to the users of NetBIOS, but does not define the way in which the protocol must be implemented. It relies on the underlying network protocol for transferring data over the wire. NetBIOS belongs in the session layer as shown in this Figure 6. NetBIOS over TCP/IP provides the means to extend the reach of NetBIOS protocol over all IP networks, thus providing interoperability with other remote operating systems.

Apart from acting as a transport mechanism for SMB, NBT provides three primary services, namely Name service, Datagram service and Session service.

Name service (UDP port 137)

The NETBIOS namespace is flat and associates hosts with unique 16-octet long names. The first 15 bytes are specified by the user, and the last byte is a unique name used by Microsoft components to identify the type of service. Various resources in the network are identified by a process defined in NetBIOS called name registration and resolution. Name registrations are done by broadcasting the unique name in the network or by registering directly to a NetBIOS name server. Resources can then be located by querying the NetBIOS name through broadcasting or querying the Name server.

Datagram service (UDP port 138)

This service provides the ability to send UDP datagrams from one unique NetBIOS name to another. This can either be a unicast or a broadcast message.

Session service (TCP port 139)

A reliable NetBIOS TCP session is established between two NetBIOS names over the TCP port 139. A successful NetBIOS session is established once the server listening to requests on the name responds positively. On successfully establishing the session service, the client and the server negotiate and continue file sharing over the session.

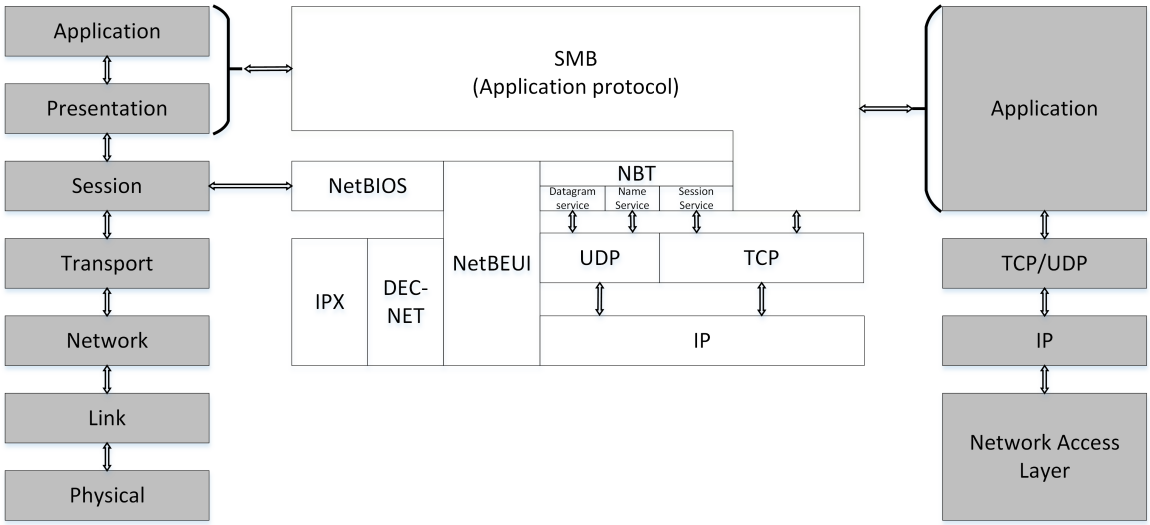


Figure 6: Layout of SMB and other transport components in the OSI and TCP/IP network model

3.1.2 Transport: Direct TCP

The new interface enables the possibility to have connections without the NetBIOS layer. The namespace used to resolve server names to IP addresses can either be from the NetBIOS name-space or the DNS name-space. The file server listening on TCP port 445 accepts the connection over which further SMB communication is processed.

The Direct TCP transport packet header has the structure shown in Figure 7.

3.1.3 Security: SPNEGO

Generic Security Service Application Programming Interface (GSS-API) provides a generic interface by which applications can interface against security protocols

Figure 7: Format of SMB transport header

in an implementation independent manner. This method allows SMB2 and SMB1 protocols versions[10] to establish secure sessions and is mandatory if one needs to implement a compliant server that supports the SMB protocol.

The standard interface and the corresponding C programming language bindings are defined in the RFC4178[11] and RFC2744[12]. GSS-API defines a number of procedures and services that can be broadly classified into three main categories, namely *Confidentiality, Integrity* and *Authentication*. Confidentiality ensures that only the involved entities are able to access the communicated data. Integrity ensures that the data exchanged between entities involved in the communication is not tampered with or modified before reaching the intended recipient(s). Authentication ensures that the identities of the communicating entities are confirmed.

As mentioned earlier, GSS-API only provides a mechanism for separating the application protocol from the authentication and authorisation protocols so that the two can function independently. GSS-API itself does not implement any of the security protocols, and depends on third-party libraries for this purpose. GSS-API follows a client-server authentication model, where the underlying authentication protocol generates a message, also known as security tokens, and relies on the application protocol to exchange these tokens. The application does not parse or understand these tokens. The exchange of the security tokens continues as long as one of the sides assumes completion of the authentication procedure. If the authentication procedure fails, the connection is dropped or error is reported. On successful authentication, session-specific services such as message specific signing or encryption can be made available and the server and client entities can exchange application specific data.

One of the caveats of GSS-API, is that it supports multiple authentication protocols and hence determining the right protocol is important. The three main methods that are often employed for selecting the authentication protocol are *Assertion, Application-*

[10] CIFS was extended with the specification[9] to allow support for authentication using SPNEGO
[11] https://www.ietf.org/rfc/rfc4178.txt
[12] https://www.ietf.org/rfc/rfc2744.txt

level Negotiation and *Negotiate*. *Assertion* works such that the server and client entities choose one method each and attempt authentication. This method is often employed in scenarios where both the client and server have the same authentication protocol. If the two entities attempt to authenticate using different protocols, then the authentication fails. With *Application-level Negotiation*, the client and server applications exchange application specific messages to determine the authentication protocol. This means that the application protocol must have the mechanisms in place to securely negotiate the security protocols for authentication. *Negotiate* uses the SPNEGO protocol[18] for selecting a mutually agreed upon authentication algorithm between the two applications.

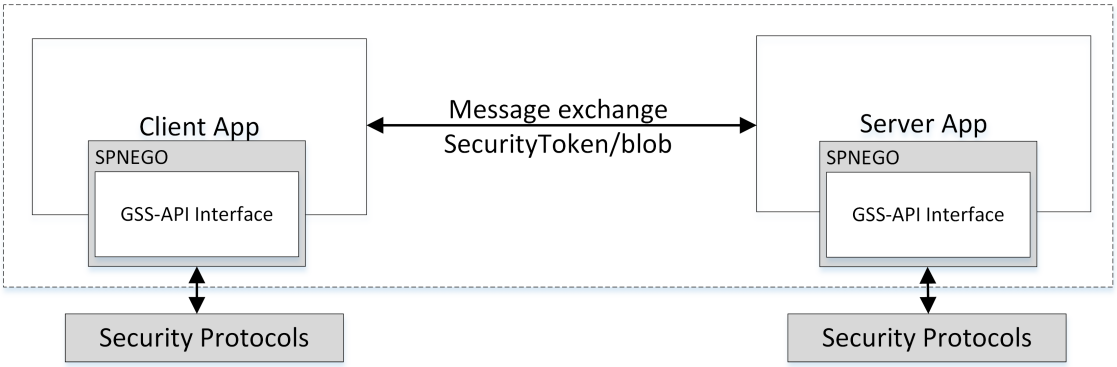


Figure 8: Authentication Model using SPNEGO as GSS-compatible wrapper for authentication protocol negotiation

Rather than creating application specific implementation for determining authentication protocol, SMB protocol uses SPNEGO which creates a GSS compatible wrapper to negotiate security protocols. Figure 8 represents how SPNEGO interacts with GSS-API and separates the application from lower level authentication protocols. SPNEGO relies on at least one of the following GSS-compatible authentication protocols in order to initiate an authentication procedure:

1. Kerberos Network authentication service defined in RFC4120[13] and technical document[19].
2. NT-LAN manager authentication protocol, NTLMSSP[20].

3.2 Packet format

A general layout of the SMB protocol packet is illustrated in Figure 9. SMB protocol implemented using direct TCP encapsulates its requests in a transport header using a format shown in Figure 7. The transport header resembles that of the NetBIOS

[13] http://www.rfc-editor.org/rfc/rfc4120.txt

session service header, and is used for representing the total combined size of the SMB requests that follow the header. The first octet of the transport header MUST be zero, followed by **StreamProtocolLength** which represents the length, in bytes, of the SMB message in network-byte order, and does not include the length of the transport packet header. Following the transport header, is the **SMB/SMB2 Message(variable)** which is a variable the SMB protocol requests for, and its length varies based on the type of command. We refer to an SMB packet as a combination of transport header and variable SMB requests.

From here on, any reference made to the term *request* is also synonymous to the term *response* unless explicitly stated.

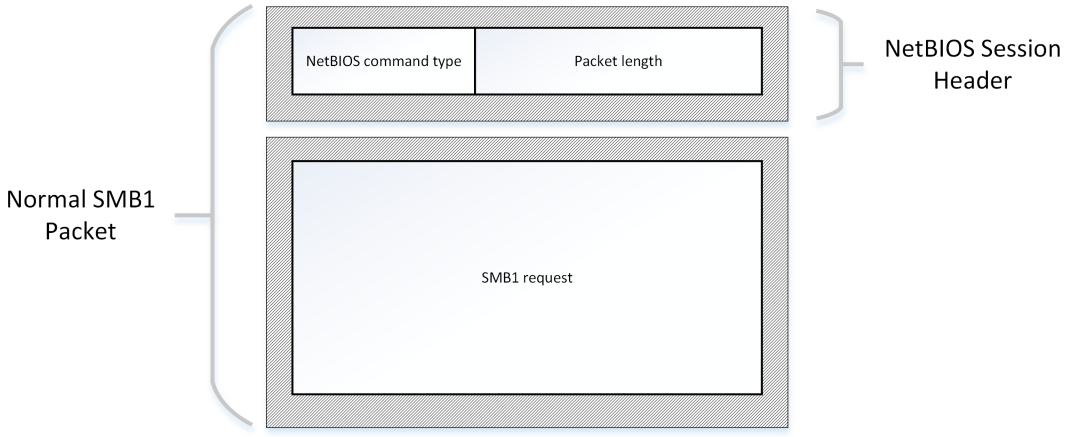


Figure 9: Layout of a Normal SMB1 packet with a single request

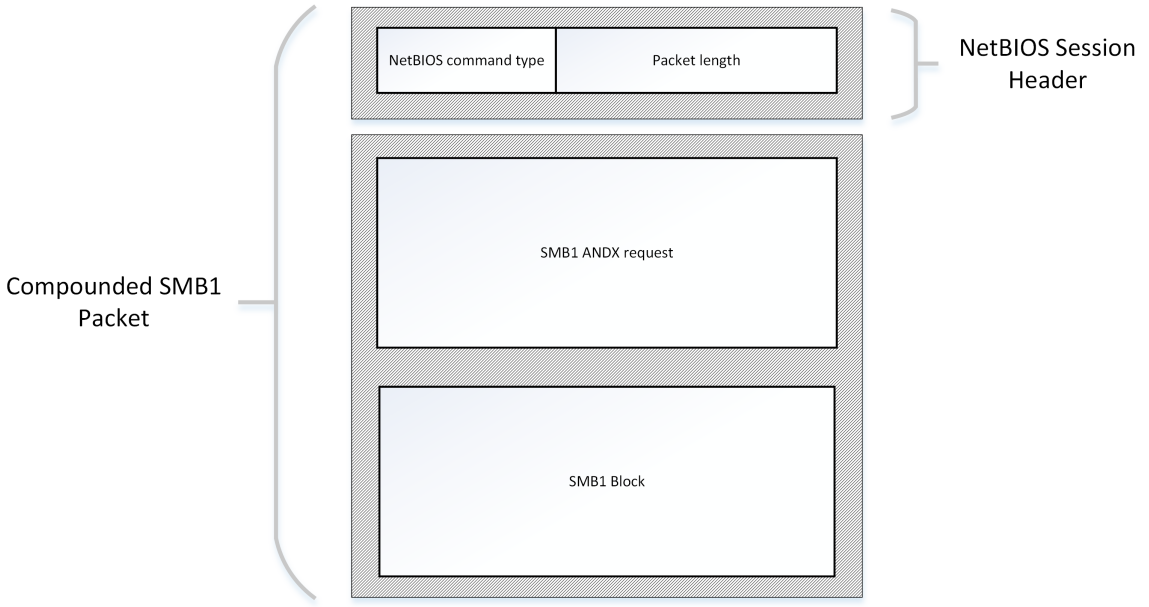


Figure 10: Layout of a Compounded SMB1 packet with multiple requests

Following the transport header, is the request corresponding to the protocol version as shown in Figures 9, 10 and 11. As can be seen from the Packet layout, both SMB1 and SMB2 packets use a single NetBIOS session service header to determine the length of a SMB protocol packet. One interesting thing to note however, is that the requests in the SMB protocol can either be single or *compounded* within a single packet. Each request in a SMB packet represents a command supported by the SMB server which will be made clear in section 3.3. *Request Compounding* means that multiple requests can be combined together in a single SMB packet in order to pipeline multiple tasks at the server and avoid round trips for performing simple tasks.

Before explaining the concept of compounding, it is important to understand that requests or commands in SMB1 protocol version can be categorized into two, namely *ANDX requests* and *Normal requests*. In SMB1, only ANDX requests can be compounded, with one exception that the last request in the compounded packet can either be a normal request or an ANDX request. In SMB2, all requests are treated equally and any number of *related* requests can be compounded, unlike the constraints put in SMB1 protocol. During our tests with compounded requests, it was revealed that the support for SMB1 compounded packets hasn't been defined in Windows operating system implementation and that they do not follow the specification. However, SMB2 compounding works exactly as defined in the specifications. The constraints on the types of requests that can be compounded for SMB1 can be found in these technical documents [8, 9].

Figure 11: Layout of a SMB2 packet with single/multiple requests

3.2.1 Request layout

The formats of the requests are completely different for SMB1 and SMB2 protocol versions. The SMB1 request layout as shown in Figure 12, has a fixed 32-byte header with some exceptions on some of the SMB1 responses such as that of the COM_READ_RAW command[8]. Following the fixed length header, is a variable length block which is comprised of the parameter block and the data block. The content of both the blocks are dependent on the command for which the request is made. Figure 10 illustrates that in a compounded packet, the first request is always a complete request, and consists of both the header and the block components. But the request that follows in the same packet, only consists of the block component. Hence processing compounded packets for SMB1 is slightly different from processing single packets, thereby introducing an additional overhead. We talk more about it later in this subsection.

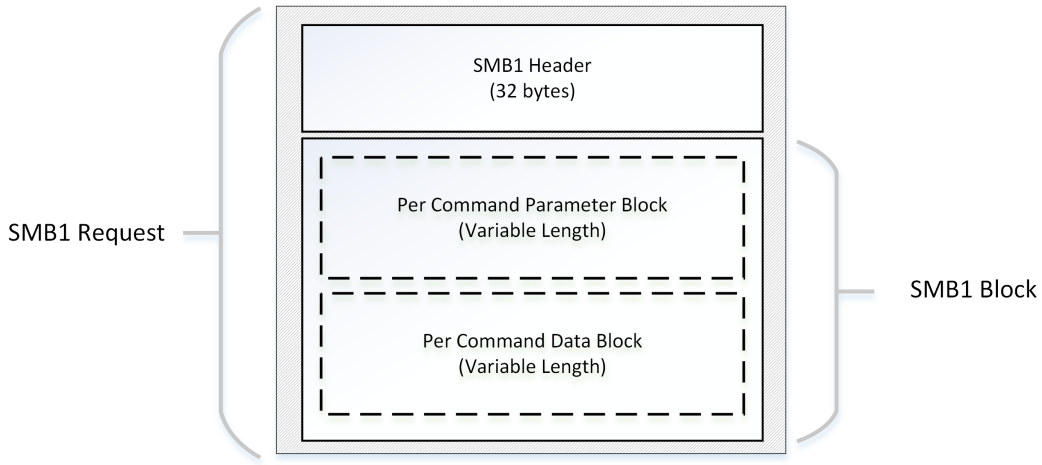


Figure 12: Layout of a SMB1 request

As illustrated in Figure 13, SMB2 on the other hand has a fixed length 64-byte header followed by a fixed length command specific structure and a variable length buffer. This buffer though, is an optional entity based on the command being processed. Most of the packet layouts for SMB2 are designed in such a way that the fields and central structures are always properly aligned. In contrast to SMB1, each of the compounded SMB2 requests within a packet have their own headers. The main advantage of this aspect of SMB2 is that the outcome of the previous request can be used to modify the header of the next request within the compounded packet. Therefore, the SMB2 requests can be pipelined. But in SMB1, this particular behavior does not exist, and hence the concept of pipelining is sort of broken in SMB1, with regard to compounded messages.

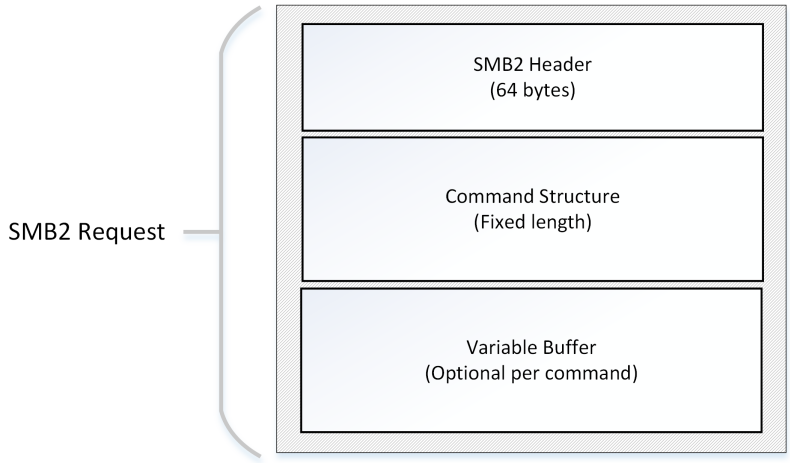


Figure 13: Layout of a SMB2 request

3.2.2 Header layout

In any SMB request following the transport NetBIOS session service header, is the version specific protocol header. Figure 14 illustrates the header of SMB1 requests and Figure 15 that of SMB2. The only resemblance between these headers is the first 4-bytes of the header which represent a magic identifier for identifying the corresponding protocol version header. The magic identifiers are:

1. For SMB1 = 0xFF, 'S', 'M', 'B' and
2. For SMB2 = 0xFE, 'S', 'M', 'B'

One of the interesting aspects of the SMB1 request header layout, is the **Command** field which identifies the command type of the current request. As mentioned earlier, in case of compounded requests for SMB1, only *ANDX* requests can be used for compounding, with the exception of the last request in the compounded packet. We have also mentioned, that for SMB1 with the exception of the first compounded request, subsequent requests do not have a request header but only an SMB1 block pair as shown in Figure 10. When messages are compounded for SMB1, it is important to identify where the next command starts. For this purpose, an ANDX command has an ANDX header in its command specific parameter block. The ANDX header has an **AndXOffset** and an **AndXCommand** field, which represent the offset of the next command and the type of command the next SMB1 block pair will represent, respectively. The end of a compounded chain of requests in a single SMB1 packet or message can be represented either by using a non-ANDX type message as the last request, or by using *0xFF* as the value of **AndXCommand** field in the ANDX header. Though the technical document does not define this aspect, as a part of the implementation we have assumed that the **AndXoffset** field always specifies an 8-byte aligned offset from the beginning of the current request to the start of the

next one. The unnecessary bytes between the end of current request and the start of the next one are zero padded.

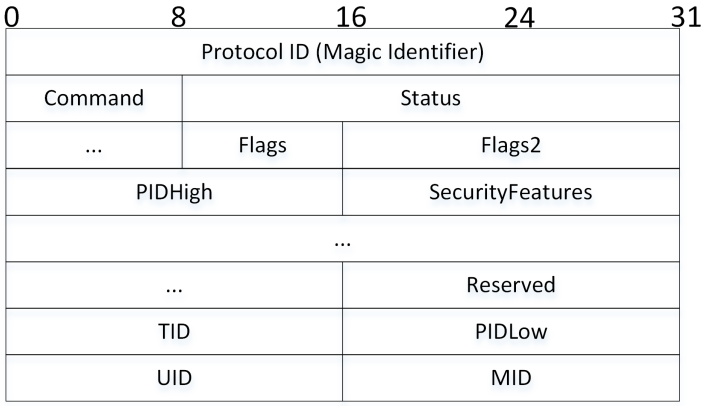


Figure 14: Layout of a SMB1 header

The SMB2 request header as illustrated in Figure 15, can come in two formats, namely *SYNC* and *ASYNC*. *SYNC* headers are used for synchronous requests where the server is expected to complete processing the request and then send the corresponding response before handling the next request. *ASYNC* headers are used for asynchronous request where the server can choose to asynchronously handle processing the request and send a response. This is especially useful if an operation gets blocked or takes a very long time. In that case, the server can choose to send an interim response for the current request it is handling, and choose to handle other requests until the activity invoked by the previous request completes. The format of the header is represented by a bit set in the **Flags** field of the header.

As already illustrated in Figure 11, unlike SMB1 which only has one header per message, compounded SMB2 requests in a message have their own headers. If the **flags** field of the header has RELATED_OPERATIONS bit set, then the next immediate request is related to the current request. This is especially useful when the outcome of one request is needed to be used for the other. For example, let us assume a case where the client wants to open a file, write to a file and close the file immediately. SMB2 can compound the open, write and close requests in a single packet with the RELATED_OPERATIONS field set. This means that the open file ID of the first request can be used to write data available in the second request, and then close the file ID from the previous request in a compounded packet. Synonymous to SMB1, SMB2 header's **NextCommand** field represents an 8-byte aligned offset of the next command which represents the offset from the beginning of the current request to the start of the next one.

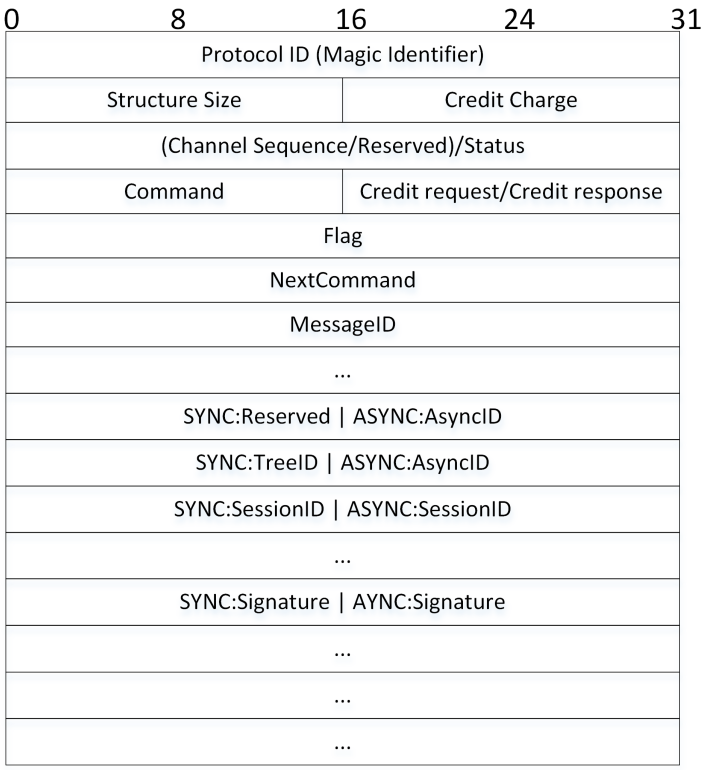


Figure 15: Layout of a SMB2 header

3.2.3 Text encoding

In SMB1, all string based fields can either be in unicode which is in UTF-16 encoding, or in an extended ascii format. All the string based fields in SMB1 are null terminated. But in the SMB server implementation, which will be discussed in the later chapters, we have enforced support for only unicode in UTF-16 encoding. This makes the implementation and protocol processing simpler, since SMB2 text encoding enforces only unicode in UTF-16 encoding. The only difference between SMB2 and SMB1 using unicode, is that in SMB2 most of the string based fields are not null terminated unless explicitly specified.

3.3 SMB commands

SMB protocol follows a client server model, and the messages exchanged between them can be broadly classified as Session control, Files/Directories access and General messages. The session control messages are responsible for authenticating, connecting and disconnecting access/connection to a shared server resource. The files and directories access messages enable access to modify files or directories on the shared server resources. The general messages pertaining to operations require

sending data to named pipes, mailslots, print queues, cache coherency and encryption.

Some scenarios demand the use of file/directory messages for general purpose messaging. Under these exceptional scenarios, instead of performing IO operations on files or directories, they are done on named pipes.

A number of clients and servers support many legacy implementations of SMB protocol versions, but since the scope of the thesis is confined to only CIFS/SMB1 and SMB2, this section will describe the various commands supported by the SMB protocol, that are required to have a specification-compliant SMB server.

Each version of the SMB protocol has its own set of commands. CIFS has over seventy five different commands, and some of the commands have their own sub set of commands, thereby making the total count of commands to be supported, to more than a hundred. With the introduction of the extension for CIFS, as defined in the technical document[9] for SMB1, most of these commands were made obsolete. SMB2, on the other hand has only 19 commands that make up the entire SMB protocol.

During the time of writing this thesis, the in-house implementation of SMB protocol that will be discussed in Chapter 4, supports 29 of the most essential SMB1 commands. Most of the other commands have been ignored because they are mostly obsolete and are not mandatory for the functioning of the SMB1 protocol version. Each of these commands have their own corresponding protocol structure that occupy the SMB1 block pair as shown in Figure 12. The Table 2, shows the list of commands that are supported in the in-house SMB1 protocol version. The first column is used to categorize commands for ease of understanding, the second column represents the textual references used for uniquely identifying the SMB1 commands, and the third column is the unique 8-bit identifier used in the **Command** field of the SMB1 header shown in Figure 14.

The list of 19 commands supported by SMB2 is shown in Table 3. Each of the commands have their own unique protocol frame present in the fixed length command structure, as illustrated in Figure 13. The last column of the Table 3, shows the unique identifier used in the 16-bit **Command** field of the SMB2 header, as shown in Figure 15. Each of the command specific structures begin with a **StructureSize** field that helps identify variable buffers towards the end of the command structure.

It is important to note, that for both SMB1 and SMB2 protocol versions, the same command can have the same or unique formats based on whether the command is a request from the client or a response from the server. SMB protocol is a request driven protocol, wherein a client sends a request and expects an appropriate reply from the server. There are however, certain messages like the CANCEL request sent from the client to the server, to cancel an outstanding request, where the client

Table 2: List of SMB1 commands and corresponding identifiers

Message Category	**Command reference**	**Identifier**
Session Control	SMB_COM_NEGOTIATE	0x72
	SMB_COM_SESSION_SETUP_ANDX	0x73
	SMB_COM_TREE_CONNECT_ANDX	0x75
	SMB_COM_TREE_DISCONNECT	0x71
	SMB_COM_LOGOFF_ANDX	0x74
	SMB_COM_PROCESS_EXIT	0x11
	SMB_COM_ECHO	0x2B
General	SMB_COM_OPEN_ANDX	0x2D
	SMB_COM_NT_CREATE_ANDX	0xA2
	SMB_COM_CREATE_DIRECTORY	0x00
	SMB_COM_READ_ANDX	0x2E
	SMB_COM_WRITE_ANDX	0x2F
	SMB_COM_FLUSH	0x05
	SMB_COM_CLOSE	0x04
	SMB_COM_LOCKING_ANDX	0x24
	SMB_COM_NT_RENAME	0xA5
	SMB_COM_RENAME	0x07
	SMB_COM_DELETE	0x06
	SMB_COM_DELETE_DIRECTORY	0x01
	SMB_COM_CHECK_DIRECTORY	0x10
	SMB_COM_FIND_CLOSE2	0x34
	SMB_COM_QUERY_INFORMATION	0x08
	SMB_COM_SET_INFORMATION	0x09
	SMB_COM_SET_INFORMATION2	0x22
	SMB_COM_TRANSACTION2	0x32
	SMB_COM_TRANSACTION	0x25
	SMB_COM_NT_TRANSACT	0xA0
	SMB_COM_CREATE_TEMPORARY	0x0E
	SMB_COM_CLOSE_PRINT_FILE	0xC2

does not expect any reply from the server. Similarly, the server can invoke a request for breaking an oplock using OPLOCK_BREAK message, which deviates from the normal client request to server response model.

Table 3: List of SMB2 commands and corresponding identifiers

Message Category	**Command reference**	**Identifier**
Session Control	SMB2_NEGOTIATE	0x0000
	SMB2_SESSION_SETUP	0x0001
	SMB2_LOGOFF	0x0002
	SMB2_TREE_CONNECT	0x0003
	SMB2_TREE_DISCONNECT	0x0004
	SMB2_ECHO	0x000D
General	SMB2_CREATE	0x0005
	SMB2_READ	0x0008
	SMB2_WRITE	0x0009
	SMB2_FLUSH	0x0007
	SMB2_CLOSE	0x0006
	SMB2_LOCK	0x000A
	SMB2_CANCEL	0x000C
	SMB2_IOCTL	0x000B
	SMB2_QUERY_DIRECTORY	0x000E
	SMB2_QUERY_INFO	0x0010
	SMB2_SET_INFO	0x0011
	SMB2_CHANGE_NOTIFY	0x000F
	SMB2_OPLOCK_BREAK	0x0012

3.4 Communication scenarios

This section covers some of the most important messages or commands that are vital for the client to establish a connection to the server successfully, as well as for it to access the shared resource from the server and other important packet exchange scenarios. There are a myriad of scenarios where different commands can be used, but this section attempts to capture the most basic scenarios for the sake of understanding how the protocol works.

3.4.1 Dialect Negotiation

After the client successfully establishes a transport connection with the server using Direct TCP, a negotiate message is sent over the newly established transport connection. Primarily, the negotiate message is exchanged between the client and server to identify the common dialect supported by the server. According to the technical specifications, the server always chooses the latest or highest supported

dialect. This is achieved by using the SMB_COM_NEGOTIATE command in SMB1 and SMB2_NEGOTIATE in SMB2.

In the case of SMB1, the client first sends a negotiate request with an array of dialect strings while for SMB2, it is an array of 16-bit integers to communicate all the supported dialects by the client. The server selects the dialect it supports from the list shared by the client, and sends a corresponding response to let the client know of its selection. It is worth noting, that clients can also negotiate newer SMB2 dialects using the older SMB1 negotiate commands. This allows for a multi-protocol exchange. The negotiate response from the server is also used for communicating various capabilities of the server. Both SMB1 and SMB2 protocol version share many features such as having a large MTU, and encryption, which can be optionally implemented by the server. This allows the clients to behave according to the features supported by the server.

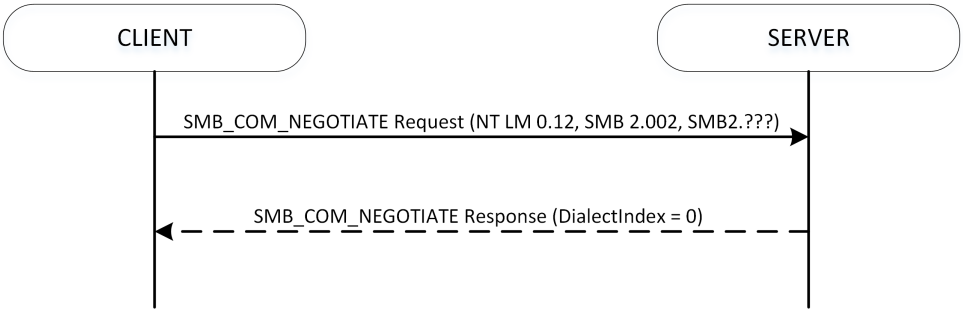


Figure 16: SMB1 Protocol Negotiation

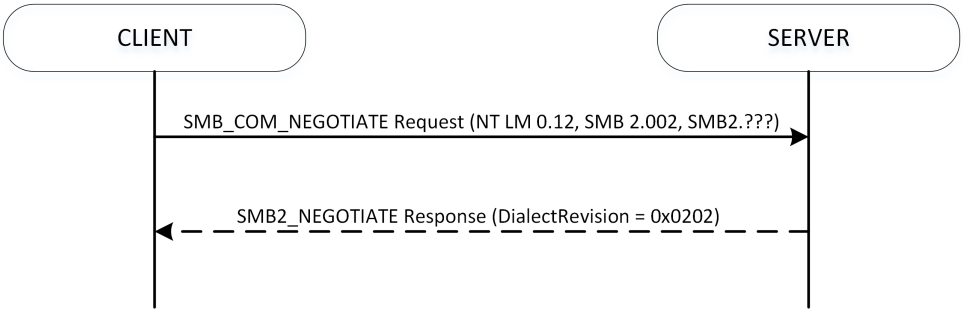


Figure 17: SMB2 MultiProtocol Negotiation when server supports SMB 2.0.2 dialect

Let us assume that the client sends a negotiate request with the older SMB1 packet format with the array of dialect strings set to **"NT LM 0.12"**, **"SMB 2.002"** and **"SMB 2.???"**. If the server supports only SMB1 protocol version, the dialect selection for further communication is done by sending the client an index of the appropriate array string using an SMB1 response. The negotiation phase follows the model as illustrated in Figure 16. In case the server supports only the base version

of the SMB2 protocol i.e. 2.0.2 (refer Table 1), it sends a 16-bit dialect identifier using SMB2 negotiate response as shown in Figure 17.

If the client supports higher dialects of the SMB2 protocol, and uses SMB1 protocol format to negotiate, it communicates the availability of higher SMB2 dialects using the **"SMB2.???"** string in its array of strings. If the server also supports higher SMB2 dialects, it responds by specifying the **"Dialect Revision"** field of the SMB2 negotiate message with **0x02FF**. The SMB client on receiving this response understands that the server also supports higher dialects of SMB2 and responds with another SMB2 negotiate request which contains an array of 16-bit SMB2 dialect identifiers. The server selects the appropriate dialect and notifies the client using the dialect identifier that the selection has been made. This 4-way negotiate handshake is illustrated in Figure 18.

It may be noted that the negotiate command is the only exception where the older command format can be used to negotiate with newer dialects. Once the server switches to the latest SMB2 version after negotiation phase, if the server receives an older protocol frame over the transport, the connection to the client is immediately terminated. If the server receives any other command before the first negotiate command, for example, the client attempting to send any sort of command before agreeing upon a common dialect, it results in termination of the connection.

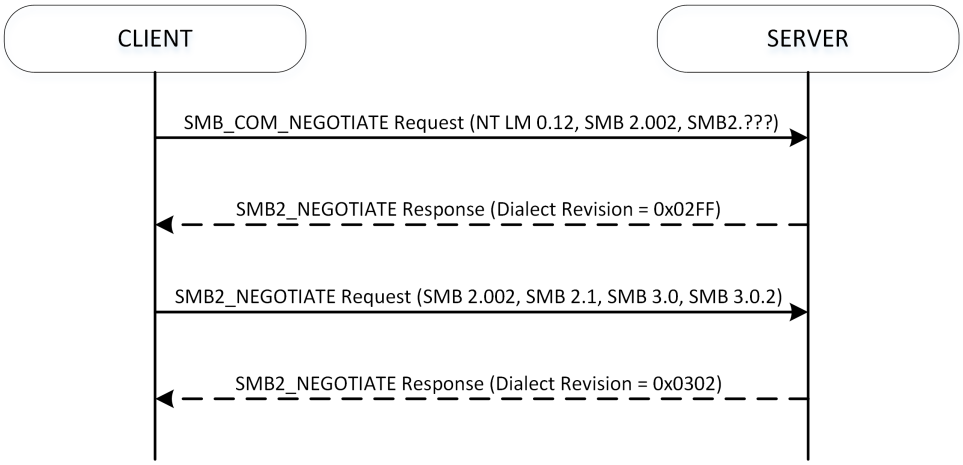


Figure 18: SMB2 MultiProtocol Negotiation when server supports SMB 3.0.2 dialect

3.4.2 User authentication

The next phase after a successful dialect negotiation, is user authentication. This is done using the session setup message. The command responsible for performing the

session setup is SMB_COM_SESSION_SETUP_ANDX in SMB1, and SMB2_-SESSION_SETUP in case of SMB2. As discussed in Section 3.1.3, SMB uses SPNEGO for user authentication. The session setup phase in SMB1 is used for sharing the capabilities of the client and these capabilities affect the way the server handles the client. Except for this oddity, both SMB1 and SMB2 protocols follow a very similar style for session setup. SMB heavily relies on SPNEGO wrapped around GSS-API as a mechanism for establishing a secure session between the client and the server. As discussed earlier, the SPNEGO negotiating mechanism chiefly uses either NTLMSSP or kerberos for authentication, though the security protocol selection process is completely hidden from both server and the client. Due to the reduced scope of this thesis, the in-kernel SMB server discussed in Chapter 4 only uses NTLM authentication. The over-all authentication process can be completed using 4 steps as shown in Figure 19.

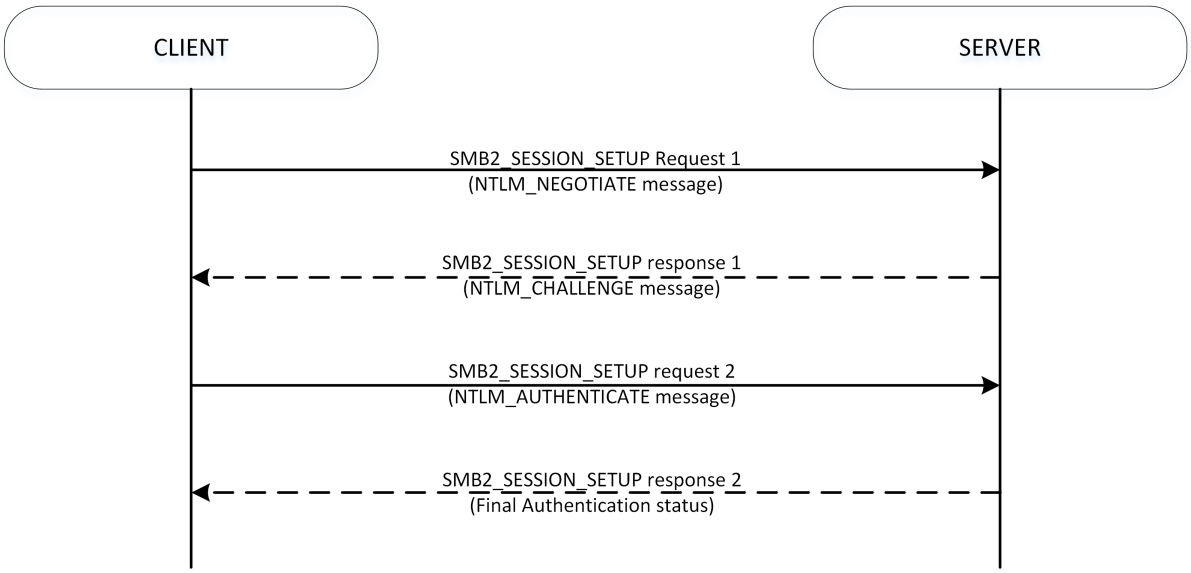


Figure 19: SMB2 Session Setup

The first step in the authentication process is the NTLM Negotiate message. This *NTLM Type 1* message is sent from the client to the server in order to initiate the NTLM authentication procedure. It is primarily used for setting the options supported via the flags field in the NTLM negotiate message. The client also sends its workstation name and the domain name which it currently is a member of.

In the second step, the server sends an NTLM Challenge message, also known as *NTLM Type 2* message, as a response to the type 1 message sent by the client. This message finalizes the options negotiated in the NTLM negotiate message and provides a challenge to the client, which is usually an 8-byte blob of random data.

In the third stage of authentication, the client sends an NTLM authenticate message. This *Type 3* message is sent in response to the server's challenge, and contains a

security blob which is generated from the user's password in response to the challenge provided in the second step. The process of generating this response is out of scope of this thesis. The type 3 message helps identify that the client has the correct password without actually sending the password to the server and also provides the username of the the account/user being authenticated.

Finally, after the NTLM 3-step authentication has been completed, if the challenge response sent in the previous message is acceptable by the server, the server successfully sends a final session setup response. Otherwise, the connection is terminated.

Both SMB1 and SMB2 follow the above mentioned steps with the only difference being that the packet formats are specific to that of the protocol version. Once the server accepts the challenge/response authentication procedure, each authenticated user is assigned a unique UID/SessionID which is returned in the final session setup response. This is used to identify the session created by the user. After successful authentication process the user is categorized as either an anonymous user, guest or an authenticated user. This is denoted in the final session setup response.

3.4.3 Connecting to a share on a server

The term **share** is used to refer to a remote resource on the server made available over the network. The method of connecting an authenticated client to a share that is exported by a server is referred to as *tree connect*. There can be multiple tree connects in a single session. This implies that an authenticated client is attempting to access multiple shares that may be exported by the server. The tree connect is done using SMB_COM_TREE_CONNECT_ANDX command in SMB1, and SMB2_TREE_CONNECT in SMB2 protocol.

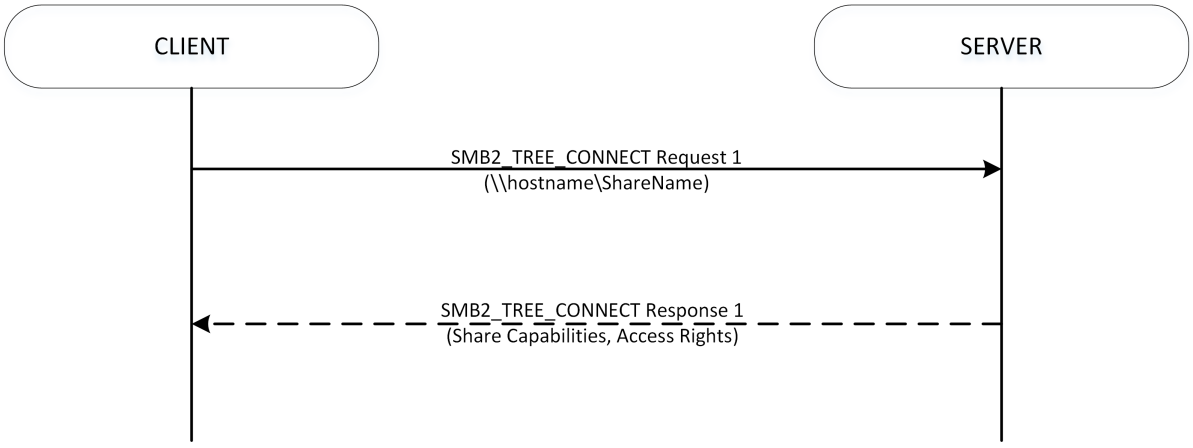


Figure 20: SMB2 Tree Connect

It is a given that the protocol frames for SMB1 and SMB2 tree connect messages are

different, but certain aspects of the commands are similar. The client invokes a tree connect request by providing a path which specifies the location of the shared device using the Universal Naming Convention (UNC) format[21]. The path must be in the format **\\hostname\share-name**, where hostname must be a NetBIOS name or a Fully Qualified Domain Name (FQDN) [14] [15] or an IPV4 address or an IPV6 address and share-name is the name of the shared resource to be accessed. Once the server receives the path and the share-name, it verifies that the server indeed exports the resources.

If the server successfully identifies the share, the response contains a unique Tree ID, that is used to identify the connection the authenticated user has on the shared resource. The response also has other fields such as: 1) Type of share i.e. if it is a disk, named-pipe or a printer, 2) Share capabilities such as caching capabilities, oplocks, etc. and 3) Maximal access rights that the user has on the currently connected shared resource. A simple tree connect exchange for SMB2 protocol is illustrated in the Figure 20.

In most scenarios, the user has no prior knowledge of the shared resource that it might want to access. In these cases, the client can query the server for a list of all the shared resources. This is done by performing a tree connect to a special service (named-pipe) called **IPC$** shared resource, and subsequently requesting for the list of shares. This action is also known as enumeration of network shares on the server. Due to the complex nature of the packet exchange, and to keep the discussion simple, this scenario is not covered in detail in this section.

3.4.4 Disconnecting a share and logging off a user

After the client completes the intended operations on a share, the final steps are to disconnect the shared resource and log off the user from the server. This is illustrated in Figure 21. The user disconnects the share using SMB_COM_TREE_DISCONNECT in SMB1, and SMB2_TREE_DISCONNECT in SMB2. The relevant field in this request is the Tree ID in the command header. The server, on receiving this request, releases all relevant resources that were opened by the client for the particular share.

Once the share has been released, the server sends a tree disconnect response with the status of the operation. The client now invokes the log off operation for disconnecting the share with the server, and also for terminating the transport connection. This is done using the SMB_COM_LOGOFF_ANDX in SMB1, and SMB2_LOGOFF in SMB2. This message does not have any relevant field in the command structure

[14] https://www.ietf.org/rfc/rfc1035.txt
[15] https://www.ietf.org/rfc/rfc1123.txt

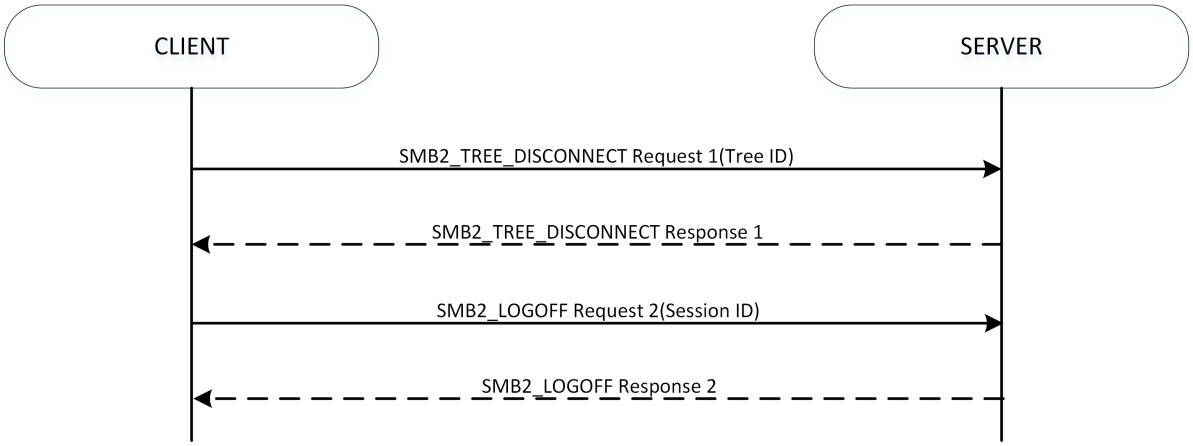


Figure 21: SMB2 Share disconnect and user log off

of the corresponding protocol frame. The only relevant field is the Session ID in the message header. On seeing the log-off command request, the server releases all session resources and sends a session logoff response before terminating the transport connection relevant for that session.

This subsection only covers some of the most basic actions that are relevant for the SMB protocol to access the shared resource. Most of the scenarios are captured as a part of the technical documentations [8, 9, 10], and is left to the discretion of the reader for further reference.

3.5 Existing Implementations

With the growing popularity of SMB as a remote file sharing protocol, and with Microsoft spear heading the development of the protocol by providing well-defined documentations, it has become a de-facto standard for file sharing across different platforms and operating systems. During the time of writing this thesis, some of the most prevalent names in the network storage industry that use SMB as one of the file transfer protocols are[16] Apple, EMC, Microsoft, NetAPP, Samba and Linux CIFS kernel module.

Apple

As of Mac OS X (10.11.5) El Capitan, Apple uses SMB 3.0 as their default NAS protocol. Apple provides its own implementation of file transfer protocol known as Apple Filing Protocol(AFP). After the release of OS X 10.9 Maverick, SMB was made the default file transfer protocol.

[16] SNIA, http://www.snia.org/sites/default/files/tutorials/FAST2016/FAST2016-TomTalpey_SMB3_Remote_File_Protocol.pdf

EMC

They provide OneFS operating system which is a scale-out NAS storage solution, and supports SMB 3.0 (default) in their 8.0 version of the operating system. Older protocol versions are supported for backward compatibility.

Microsoft

Being the main contributor to the SMB technical specification, they support SMB 3.1.1(default) as of Windows 10 and Windows server 2016 preview. Older versions of the operating system implement older SMB protocol version and dialect for backward compatibility.

NetApp

They use data ONTAP 9.0 series operating system for implementing a hybrid file-system for acting both as a NFS and SMB server. It uses SMB 3.0(default).

Samba

In this list of available servers, Samba is the only open source user-space implementation that is most widely used both as a server and a client in a Linux based operating system. SMB protocol version support for SMB 3.1.1 is available as of Samba version 4.4.5.

CIFS

CIFS is the open source kernel-space implementation of SMB protocol client. CIFS client has support for SMB 3.0.2 as of CIFS version 2.09 released in Linux kernel 4.7.

3.5.1 Samba: Network file system for Linux

Linux is not only designed as an all purpose operating system, but also for more specialized areas of different scales including embedded platforms, enterprise platforms or for real-time applications. With Linux becoming one of the major players in the networked ecosystem of mobile devices, routers, large server farms, etc. and with many other operating systems following suit, it is clear from the discussion in section 3.5 that one of the most common file sharing solution implemented is the SMB protocol. We are especially interested in the SMB implementation available in Linux. CIFS already serves as an excellent in-kernel SMB client alternative to Samba's own user-space client, namely smbclient. Samba provides a free software suite in user-space for providing file sharing and printing services by using the SMB protocol. This is the only available non-commercial SMB server. Many embedded devices that appear in the market such as NAS boxes, routers, TV and Set-Top Boxes(STB) and

Set-Top Units (STU), either have a server for hosting files or a client that is capable of accessing files hosted by other devices.

It is a general belief that user-mode servers perform generally worse than their counterparts implemented in the kernel space. Initial research in evaluating the performance of user-mode and kernel mode web servers have shown that there is a significant gap in performance between the two modes of the web servers[24, 25]. The research shows that under a plethora of different dynamic and static content workloads, the kernel mode is definitely better compared to its user-mode counterpart. Even though web servers are significantly different from network file sharing servers, it can be assumed that both of them are network applications and hence the same argument can be applied to the latter. One of the most interesting aspects of the initial research, is that the kernel web servers perform significantly better than user mode web servers with respect to static content. This study is relevant to the SMB protocol, since most of the content transferred between the client and the server are static.

The open source Samba project presents a number of limitations, and has not been tailored for use on an embedded device. Another major limitation of the Samba is that from version 3.1.X and higher, it follows the GPLv3 licensing, which could make it legally tricky to be used on commercial products.

3.5.2 User-mode server limitations

Before attempting to re-design an existing, fully compliant user-space network application like Samba in kernel space, it is important to understand some of the obvious problems that impact the performance of a server residing in the user space.

Boundary crossing and data copies between user and kernel

Data copy between the user-space and kernel becomes extremely hard to avoid, specially since the data usually resides on disk or in the application buffer waiting to be flushed on to the disk. Both kernel and user-space have their respective address spaces. Thus when data is being copied, say from user to kernel, the kernel copies and stores the data in its own address space. This process of copying data across the address boundaries is time consuming. This introduces a performance penalty whenever data crosses from the user-space to the kernel space boundary. Hence designing the application in a way that reduces the number of boundary crossings, can significantly boost performance. Latest Linux kernel versions export a zero-copy ability to reduce the impact of boundary crossing, thereby reducing the performance gap between kernel and user-space applications, by using the concept of *sendfile*.

Network event notification

Another important performance issue is the cost of notifying a socket or network event to the user-space server. Most often, high performance network servers are implemented with an event-driven design. In an event driven implementation, dispatch mechanism such as select() or poll() is used. The primary problem with these mechanisms, is that a single system call must notify the kernel of all the events the user application is interested in, and in turn must wait for notification from the kernel. This results in a large overhead, specially in environments with large number of connections or large number of events and short lived connections[26].

It is possible to reduce the overhead introduced due to the event sets in the select() call, by using a thread based model with very minimalistic scheduling overhead, or a multi process model where a new process is created for handling new connections. Samba has adopted the multi process method wherein each individual connection is handled by a new process. One of the drawbacks of this model, is that in a system with limited resources such as an embedded device, this will not scale and as a result the number of users will be limited. Hence an event notification mechanism with good scalability is not possible in the user-space.

Overhead of context switching

In a high performance network application such as a file sharing server, a number of system calls are made specially during a data transfer, which can saturate the CPU. Apart from data transfer scenarios, most operations involving file handling invoke a number of system calls that pass operations to the underlying file system. As a result, a context switch may be performed. Context switching can impose a severe performance penalty, although the extent of the penalty depends on different factors such as the workload, memory read/write patterns, the type of processor architectures etc.

Research[27, 28] has shown that there is severe performance penalty introduced due to context switching. Factors leading to such penalty are an invalidation of the process-relevant entries in the processor's address translation buffer (TLB), when switching address space between user and kernel, memory page swapping which involves moving pages to swap disks and flushing & reloading of processor data caches, instructions etc. Traditional server applications use sockets and their APIs, and this requires servers to invoke system calls like select(), read(), write() etc. These calls introduce a performance overhead by crossing address boundaries, and are invoked at a very high rate on servers undergoing extremely high load. This can severely limit the performance of the network stack[29].

4 Developing a File sharing solution

This chapter describes the file sharing solution that was developed for embedded and enterprise systems, while also being inter-operable with many SMB clients. An overview of the project is provided in Section 4.1. Section 4.2 gives a brief summary of various software choices made and third party libraries that were used to implement the server. Section 4.3 gives a detailed explanation on the architecture of the proposed solution. It also describes the implementation aspects of the proposed solution and how different components interact. Section 4.4 provides an overview of various features and functionalities that can be expected from the file sharing server.

4.1 Project overview

In order to overcome some of the obvious performance costs introduced by running a server in user-space, and to fulfil the needs of being inter-operable with multiple devices and platforms, Tuxera Inc kick-started a project named Tuxera Server Message Block (TSMB). Tuxera has been the market leader in creating inter-operable kernel file systems such as NTFS, exFAT and FAT in Linux and brings with it nearly 20 years of expertise in the field of creating file systems in kernel and user-space.

The primary goal of this project, is to make an in-kernel file sharing server capable of outperforming its user-space counterparts, while adhering to the requirements posed by both the embedded and enterprise systems. Most of the embedded devices use either ARM or MIPS processors, and have very low on-board RAM, low CPU frequency and a very limited on-board storage, while still attempting to provide a feature rich platform. Some of the basic requirements when running on such low end platforms, are that the server must be able to handle any load while running with a very low memory footprint and using the CPU resources fairly and efficiently. On the other hand, enterprise systems do not impose any such stringent requirement. The

primary design of the in-kernel server was built around the constraints enforced by an embedded platform. We are fairly confident that such a server can also perform well enough on any high end device.

We have been able to complete a fully compliant SMB protocol server that can work with almost any client. Some of the tested clients include Windows, CIFS & the smbclient in Linux, and clients from Mac OS. This fully compliant SMB server has already been shipped to a number of manufacturers for evaluation. Being one of the major contributors to this project, I have helped with the design and implementation of the server. Some of my development efforts involve contributing and developing the base SMB1 and SMB2 protocol stack, testing, fixing and improving POSIX compliance of the product, etc. Due to the complexity of the project, stemming from a rich collection of messages, protocols and technical documents that were used as references, the project proved to be a considerable undertaking with the base file sharing protocol framework consisting of more than 100,000 lines of code.

4.2 Software Choices

This section covers some of the vital software decisions that were made before the development of an in-kernel file sharing server.

4.2.1 Kernel

Due to the nature of development, it was important to select an operating system that could be modified freely and compiled for instrumenting, debugging, etc. Our obvious choice was the Linux kernel[17], and most of the development took place in the Ubuntu flavour of the Linux kernel. Of course, one of the major factors in determining the use of Linux was that Tuxera Inc, is mainly based on creating commercially inter-operable software for Linux. Most modern data centres have started adopting Linux as one of the mainstream server operating systems due to the ease of availability of commodity off-the-shelf Linux servers[18].

Henceforth, please note that any reference to kernel will always refer to the Linux kernel unless stated otherwise.

[17] https://www.kernel.org/

[18] https://www.redhat.com/f/pdf/IDC_Linux_Mainstream.pdf

4.2.2 Third Party Libraries

Due to the versatile nature of the project, there was a choice between creating indigenous libraries which might be a re-implementation of existing libraries, and utilizing existing third party libraries to implement and augment features and functionalities that would otherwise have been impossible to conceive in a limited time period.

Since the project uses both user-space subsystems and kernel space modules that fuse together to form a hybrid architecture, as clarified in Section 4.3, it sometimes puts restrictions on the usage of third party libraries. Some of the third party libraries that have been used are OpenSSL and DCE/RPC.

OpenSSL

OpenSSL[19] is an open source project that provides a commercial grade toolkit for TLS and SSL protocols and implements some basic cryptographic functions. Some of the necessary signing and encryption routines have already been developed at Tuxera. As explained earlier in Section 3.1.3, authentication is done via GSS-API, with the messages encoded using ASN.1[20]. Despite having implemented most of the routines locally, the in-house library variant did not have the facility to support ASN.1 encoding. As a result, we had to choose the OpenSSL library.

DCE/RPC

DCE/RPC is an implementation of the Remote Procedure Call (RPC), which forms the basis for windows RPC implementation. This library was heavily modified in-house to suit the needs of the in-kernel server architecture, with various bug-fixes and new definitions to support missing features.

4.2.3 Programming Languages

Being the de-facto coding standard in the Linux kernel and for historic reasons, **'C'** programming was the only obvious choice that guaranteed compatibility, portability and performance. Compilers for C are available for almost all hardware architectures and operating systems, making it one of the obvious choices for a cross-platform programming language.

Since the architecture in the implementation involves both user space and kernel space components, we use both standard C libraries and non standard APIs for

[19] https://www.openssl.org
[20] http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx

various kernel subsystems. When compiling kernel modules, linking libraries of various subsystems such as threading, file and network management and memory management, does not happen against the standard C libraries as in the case of user-space. Unlike *syscalls*, the linking[21] in kernel modules happens against routines exported by the the kernel subsystems. It is also possible that the same function call in the kernel may return or accept arguments differently across different versions of the kernel. This must be taken care of as well, in order to work against older and newer kernel versions.

All other support tasks like creating and invoking a test framework, analysing and generating test reports and other configuration related tasks for building the project have been done using shell scripts.

4.3 Design and Architecture

This section describes a high-level modular architectural design, and the implementation of the SMB protocol-based file sharing server, introduced in Chapters 2 and 3, as both a kernel-space server and a user-space application, for the purpose of evaluation. In the required parts, we will discuss about the important data structures used and various techniques adopted for a seamless functioning between the different user-space and kernel components. For the sake of brevity, this section will not delve into the low level technical details, unless necessary.

The server that has been implemented for discussion in this thesis was designed to run in many modes, as per the compilation procedure invoked by the user. The different modes supported by the server are shown in Table 4. The first column explains the format in which the server can be invoked, either as a user-space or as a kernel server. The second column gives info on the number of binaries that need to be invoked to get the server functioning. The third column defines the type of binary i.e. a user-space binary or a module in kernel. The fourth column describes how the binary defined in the third column invokes its subsystems, either as a thread in a process or as individual processes using the fork() Linux API. In the kernel, all individual processes are invoked as kernel threads.

If the server is compiled in a *user-space format*, the individual components of the architecture reside completely in the user-space. Since the architecture consists of multiple components or subsystems, the user-space application can be run in two different modes as shown in the fourth column of Table 4. In *multi process* mode, the server application invokes each of its subsystems or components as an individual process in the Linux environment, whereas in *single process* mode, the

[21] Linkers and Loaders, 1st edition, John R. Levine

Table 4: Format and modes supported by the TSMB server

Server Format	**No of binaries**	**Binary type**	**Server mode**
User space format	1	Server binary	Single Process (Threaded)
	1	Server binary	Multi Process (Forked)
Kernel space format	2	Server Binary	Single Process (Threaded)
			Multi Process (Forked)
		Kernel module	Multiple kernel threads

server application runs as a single process and invokes each of its components as a thread running under the main process. Figure 22 illustrates the different components in the user-space format of the server. Evaluating the user-space format of the server is out of scope of this thesis.

If the server is compiled in *kernel space format*, the resulting server is divided into two binary components. The core components of the server are compiled as a kernel module and some of the services that are essential for the server are compiled as a user-space application. To get the server functioning, the kernel module must be inserted[22] into the Linux kernel first, followed by running the user-space server binary. The design behind such an approach, is that the most performance critical components are moved to the kernel, whereas other components and services that only supplement the SMB file sharing protocol are in the user-space. The components that supplement the core SMB file sharing protocol are known as *services*. Some of the services are mandatory for the functioning of the file sharing protocol, while the others complement the file sharing server by introducing additional functionalities like printer services. Since these service components are not performance critical, a decision was made to place them in the user-space. The Figure 23 illustrates the segregation of the components across the user-space and kernel-space.

This major design decision to run the server in two different formats was made so that the user who hosts a file sharing server can choose to either run the server as an application in the user-space, or as a server thread in the kernel-space. The design was made such that there is no difference while operating and configuring the two different formats of the file sharing server.

The key components of the in-kernel SMB architecture are *Message Queuing Subsystem*, *TSMB Core*, *TSMB Authenticator Service*, *TSMB SRV service* and finally *Kernel and User-space Abstraction*.

Briefly speaking, the *Message Queuing Subsystem* describes the methodology used

[22] http://www.tldp.org/HOWTO/Module-HOWTO/x197.html

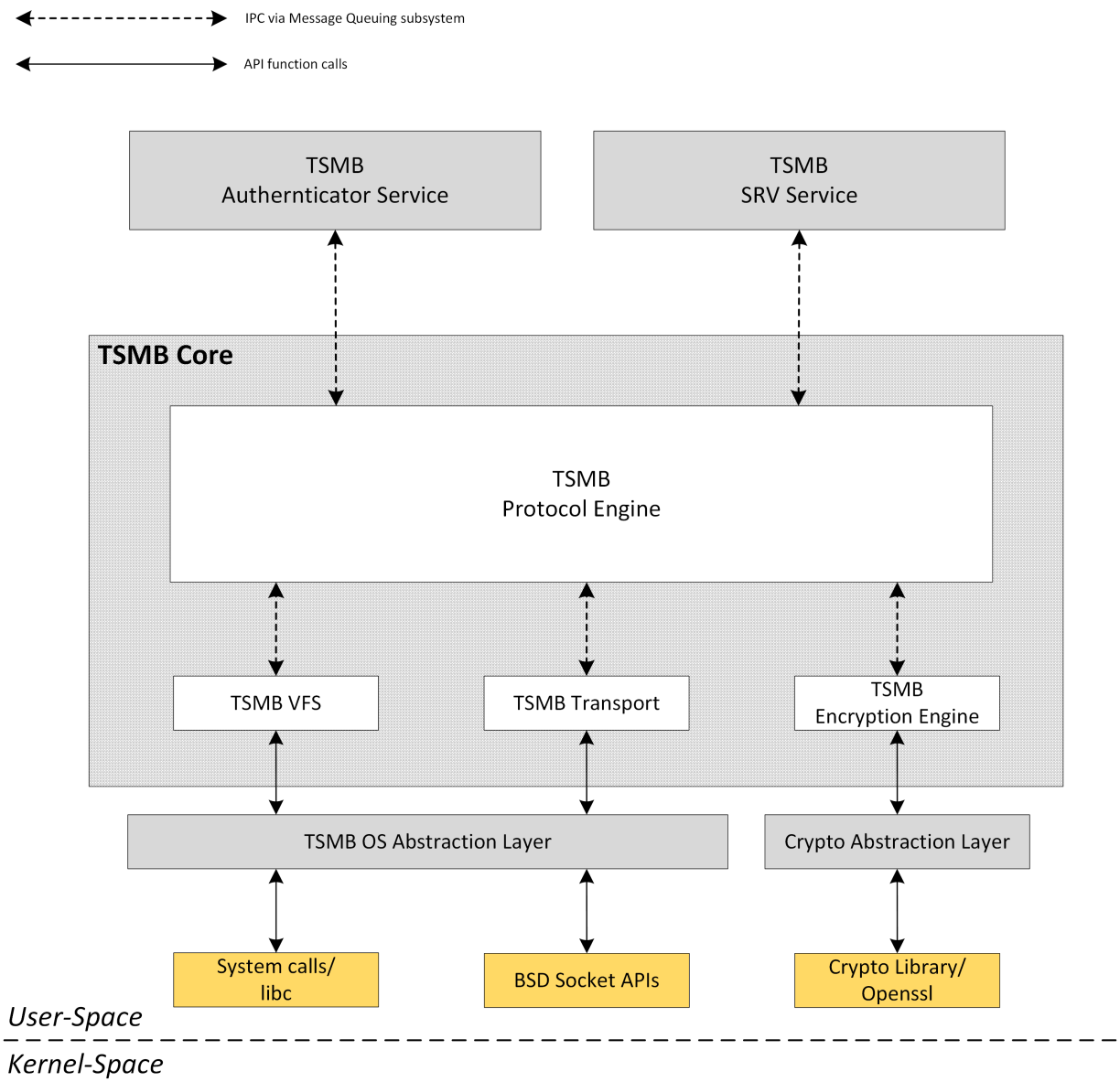


Figure 22: Simplified TSMB Architecture as a User-mode server

for inter-process communication between the different components, described in Section 4.3.1. The *TSMB Core* outlines the main component that implements all the necessary functionalities for seamless functioning of the SMB protocol as a file sharing server, described in Section 4.3.4. *TSMB Authenticator Service* introduces the authenticator service component of the TSMB architecture, that plays a vital role in authenticating the user and generating the security context of a user as described in Section 4.3.2. *TSMB SRV service* provides an overview of another service component that implements the Server Service Remote Protocol[11], as described in Section 4.3.3. *Kernel and User-space Abstraction* briefly presents the mechanisms that have been used to implement abstractions for both kernel and user-space operations, as

described in Section 4.3.5.

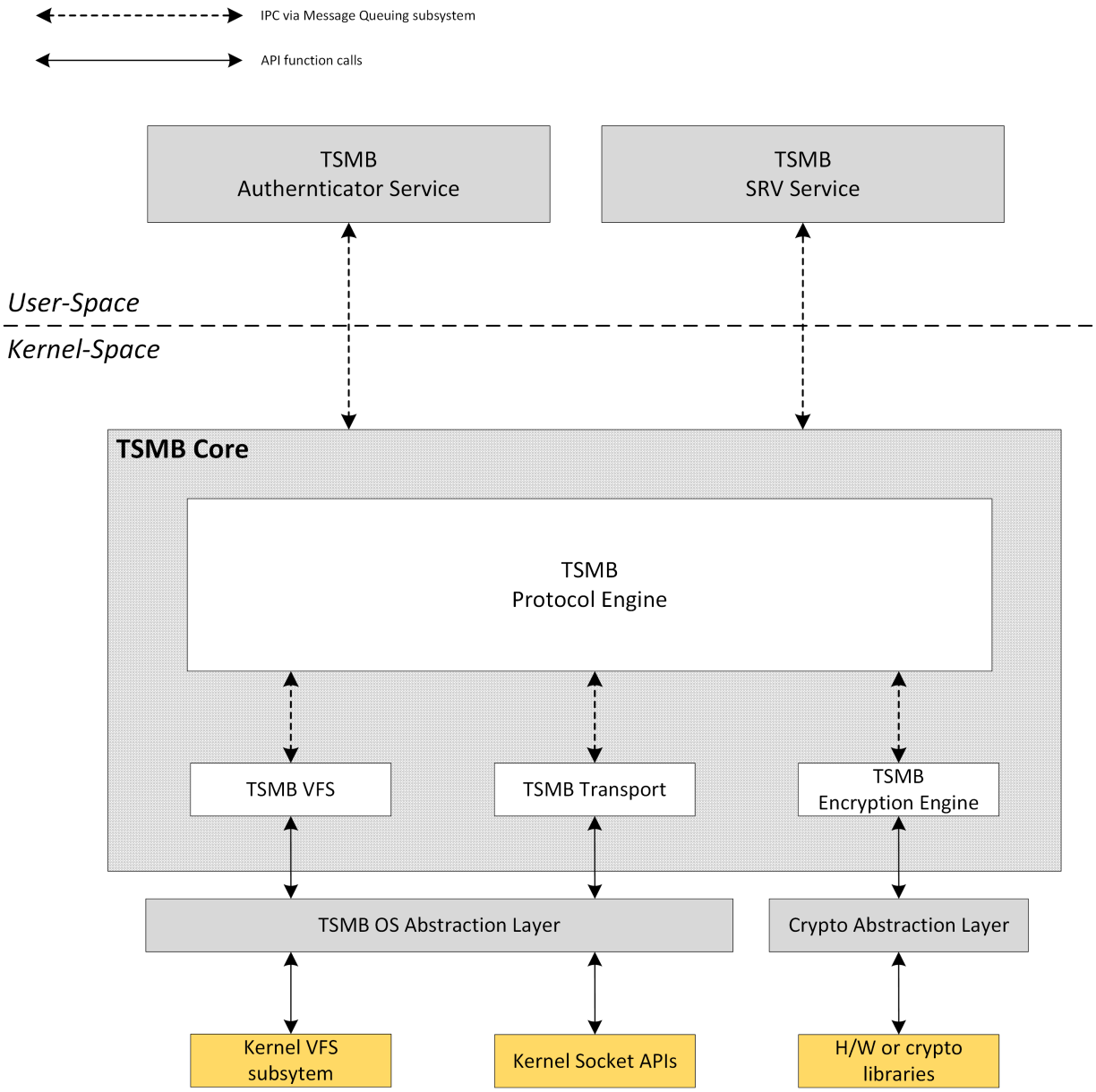


Figure 23: Simplified TSMB Architecture as a Kernel-mode server

4.3.1 Component: Message Queuing Subsystem

Though message queuing may sound as simple as a queue which accepts message blocks for processing, we refer to it as a subsystem since it forms the basis of communication across different components in the TSMB architecture. All transactions and communications between different components take place in the form of message units. Each message unit is unique and can be categorized into:

1. Management message unit
2. Protocol message unit

A management message unit is responsible for various activities involving configurations that affect the way a component behaves or the way the server behaves.

A protocol message unit usually involves parsing the message block of the SMB protocol packets, for either direct protocol processing as defined in SMB technical documents or for forwarding the message unit to other services that use SMB protocol as a transport.

Each of the components discussed in the subsequent sections have a message queue for receiving and transmitting messages, either to be consumed by the service or to be forwarded to other appropriate services for further processing.

4.3.2 Component: TSMB Authenticator service

In the server architecture, some of the less frequently used services that do not have a performance impact in the functioning of the SMB protocol have permanently been placed in the user-space, irrespective of whether the server was compiled to run in kernel or user-space format. As mentioned earlier, **services** are components that may or may not be essential for the functioning of the SMB file sharing server.

The main task of the authenticator service is to provide an authentication interface to the TSMB core. This service component is absolutely essential because without a functional authenticator service, users will be denied access to the shares hosted by the server. On successfully authenticating a user, this component generates security blobs that house a treasure cove of security information about a particular user. These security blobs include information such as User ID, list of Group IDs, which are relevant in Linux, and Security Ids, which are unique values used by Windows operating system for identifying security policies associated with a user account or group and other security related information. Although the SMB authentication guarantees support for many authenticating interface theoretically, the current implementation only uses NTLMSSP authentication protocol between a client and a server, by deriving cryptographic keys from the shared password secret.

This component is not performance critical, since the communication between the core and authenticator services can happen only when the user attempts to (re-)authenticate with the server, which is commonly a one-time thing per user per session.

4.3.3 Component: TSMB SRV service

The second essential component for the SMB file sharing server, is the SRV service, also known as the Server Service Remote Protocol[11]. This is an RPC based protocol that uses the SMB protocol as its transport for administrative tasks. For one, it can be used for remotely enabling file and printer shares on the server, and for exporting some named pipe services to the server. Besides that, it can also be used for remote administration of the SMB server such as remotely querying and configuring the server. The server can be queried for useful statistics such as number of active connections, shares, open files, etc.

In addition, the SRV service reads the configuration file required for configuring the server, which includes information such as the IP address and the port number to accept connections, the number of shares that the server should expose to remote clients, etc. The SRV service then passes this configuration information to the core component, so that the file server can start listening for connections. SRV service is also capable of dynamically adding and deleting file shares on the server during runtime, and changing the type of services running on the server. Finally, the SRV service exposes additional services to clients in the form of *endpoints*, that are exposed by DCE/RPC. One such example is the **"srvsvc"** endpoint, that uses protocol sequences for RPC over SMB (NCACN_NP)[22].

4.3.4 Component: TSMB Core

As the name implies, this component forms the core of the TSMB architecture. The core takes on the role of invoking its sub-components, that are necessary for the functioning of the file sharing server, while also primarily functioning as a SMB protocol processing engine. One of the primary differences between the user-space and kernel-space port of the SMB file sharing server, is the way this core interacts with the operating system. As shown in Figure 22 and 23, the core component can either reside in user-space or kernel-space based on the format of the server.

Transport

This sub-component is responsible for accepting new connections, handling existing ones and cleaning up allocated resources as and when a client disconnects gracefully or tears down a remote end point. It is also responsible for receiving and transmitting data packets over a valid connection. During reception, it ensures that the received packet is indeed a valid packet and then forwards the packet to the appropriate sub-component, usually the protocol engine.

This transport thread initializes and starts waiting for new connections only after receiving a configuration message from the core. This configuration message contains the address and port information, that was parsed by the SRV service.

As seen in Figure 22 and 23, the transport sub-component resides in the kernel-space or user-space, based on the format of the server. In kernel mode, since the transport resides in the kernel space it is possible to tightly integrate with kernel network stack for improving performance.

VFS

The VFS sub-component, also known as the share component, is a virtual file-system layer within the SMB file sharing server that facilitates an abstract model for implementing file/directory IO operations, as specified in the technical documents[23][23].

As mentioned earlier, the SRV service reads the configuration file for shares to be exposed to clients. On finding shares with a valid path, the configuration parameters associated with these shares are sent to the core component, which in turn processes these messages as a share registration message. On receiving a share registration message, the core component invokes a VFS thread for that share. In the current design, each share is handled by an individual VFS thread, and hence all IO operations for a particular share are handled by that thread.

By moving the VFS component to the kernel space, it is possible to directly interface with the Linux VFS layer rather than invoking syscalls for performing IO operations from the user-space, as is the case with any user space application.

Protocol Engine

The primary task of the core component is to not only invoke some of its other sub-components, but also to act as a protocol engine capable of processing SMB protocol packets. Being the heart of the core component, the protocol engine is responsible for four important tasks. Firstly, it is responsible for receiving share configuration and network configuration messages from SRV service. On receiving share configuration messages, it invokes the corresponding share threads while on receiving a network configuration message, it is transmitted to transport thread for further processing. Secondly, on accepting new network connections and performing a *session setup* for a new user, it transmits and receives messages from the Authenticator service for authenticating the user as described in Section 3.4.2. Thirdly, on receiving SMB

[23] http://download.microsoft.com/download/4/3/8/43889780-8d45-4b2e-9d3a-c696a890309f/File%20System%20Behavior%20Overview.pdf

protocol message that invokes IO operation on a share, it constructs a VFS message and sends it to the corresponding VFS thread for further processing. Lastly, on receiving an SMB protocol message that involves access to named pipe endpoints, exported by any additional service e.g. the SRV service, it constructs a service specific message and forwards it to the servicing component for further processing.

The protocol engine is optimized and is free of locks and sleeps, hence it is extremely fast when processing protocol packets. The protocol engine resides in the kernel-space when using the kernel mode. The idea here, is that it can make use of kernel primitives to further enhance processing and sending response to client issuing a protocol request. This way the engine resides very close to both the transport as well as the VFS component.

One of the fascinating aspects about the design of the SMB protocol engine, is that on receiving a valid SMB protocol packet, it is processed by either the SMB1 or SMB2 protocol sub-engines. Suppose the received request is for reading a file, at the end of processing the protocol packet, the protocol engine formulates a protocol-independent read request to the VFS component. So the only difference while processing different protocol versions, is the way the packet is parsed within the engine. This ensures that the behaviour of the server is nearly consistent across protocol versions, unless otherwise specified in their corresponding technical documents.

Encryption Engine

The encryption engine is responsible for encrypting the protocol data unit and forwarding the encoded packet to the transport thread for transmission. It can also decode encrypted packets from an authenticate user-session, and then forward the decoded packet back to the protocol engine for further processing.

The SMB 2.0 and SMB 2.1 protocol dialects do not support encryption. Optional encryption was added as part of the SMB 3.x dialect family. The cryptographic algorithms used for encryption are AES-128-CCM or AED-128-GCM, based on the cipher ID that has been negotiated. These cryptographic algorithms have been implemented as a separate library in-order to reduce the dependency on any external library.

4.3.5 Kernel and User-space Abstraction

One of the unique requirements when designing the TSMB architecture, was the need to be able to host the file sharing server in two distinct environmental scenarios, namely a kernel-module and a user-space application. The easiest way to go about doing so, would be to ideally have very minimal code modifications so that one can

ensure that there is absolutely no difference between the two. Thus, any performance comparison between the user-space and kernel space TSMB architecture remains fair.

Keeping the code changes minimal is no trivial task, since the nature of programming in both the kernel and user-space is different. Many major obstacles mar the path to developing efficient applications that can run in both the programming paradigms. Some of the most obvious difficulties faced while attempting to maintain a common abstraction between the two formats are *sanity & fault tolerance*, *debugging*, *libraries* and *documentation*.

When considering sanity and fault tolerance, it is important to understand that the nature of fault recovery is different in kernel and user-space. The kernel and the kernel module share the same address space as other sub-systems in the Linux operating system. Due to lack of any protections against corruption, caused by wrong memory access or over-writing addresses from other sub-systems, it is possible that mishandled error scenarios can cause a kernel panic leading to a crash or a system freeze. The user-space application on the other hand, is complemented by sanity checks that are introduced in the kernel space for any system call the user application makes. Since each user-space process runs in its own address space, any access to memory outside its own address-space, can cause a segmentation fault causing only the application to crash while the operating system is still functioning seamlessly.

Debugging user-space applications has been made easier, with the availability of a number of tools along with a myriad of performance evaluation and memory profiling tools, that can pin-point locations causing high CPU saturation or memory consumption. Though the kernel-space modules also have their own utilities such as o-profiler, system-tap, perf tools etc., most tools require augmenting and re-compiling the kernel with new configuration settings. These tools have a very steep learning curve, unlike its user-space counter part.

One of the advantages of programming in user-space, comes from the ease of use of standard libraries that provide standard system call APIs for file, network and memory IO management, locking, etc. Unlike user-space, the kernel-space makes use of calls to non-standard kernel routines which are exported by different subsystems. The kernel subsystems are continually updated, and as a result the nature of the API is bound to change across kernel release versions. The kernel APIs that can be invoked by kernel modules depend on whether the modules have a proprietary or General Public Licence (GPL). Some of the performance critical kernel calls are exported only to modules that hold a GPL license. Any module that has a proprietary license cannot make use of these calls, making it even more difficult to program.

An obvious issue in kernel-space programming, is the lack of proper documentation

defining all the kernel APIs. Though a number of textual references[24] [25] are available, they are constantly out-paced by the rate of development in the Linux kernel community. In contrast, all user-space APIs are clearly documented, and a very good user-manual database has been made available for development.

In order to have a smooth transition between the kernel and user-space modes of the server, an abstraction layer has been introduced to act like a software shim or a wrapper. This abstraction is designed in such a way that the underlying APIs can be switched out without the knowledge of the rest of the code. This way, the high level APIs are able to successfully unify the specifics of different modes by providing a consistent interface, behind which most of the abstraction is conveniently hidden. This selection of the abstraction layer, that is compatible with the mode of the server, is done during the compilation process of the server application.

The components that were described in the previous sections have the same routine signature i.e. there is absolutely no difference in the code used between the user-space and kernel-space components, except for the layer of abstraction. This greatly helps reduce the complexity in designing a multi-format and multi-mode server that can work both in kernel and user-space.

The abstraction layer was mainly aimed at the following operating system components:

1. Threads
2. Memory management
3. Locking
4. Timers
5. Transport or network management

For invoking threads, the user space server makes use of pthreads, whereas the kernel-space server invokes kernel threads.

The user-space server makes use of malloc and its variants provided by libc for dynamic memory management, while kernel-space server makes use of kmalloc, vmalloc, etc., for managing kernel memory.

The locking layer is abstracted in order to provide synchronization mechanisms, which includes mutual exclusion for critical sections, scheduling threads until a desired event occurs, and protecting shared data structures from simultaneous modifications. This is achieved through pthread locks and semaphores in user-space, whereas the kernel-space makes use of spinlocks, mutex, etc.

[24] Linux Kernel Development, Third edition by Robert Love
[25] Professional Linux Kernel Architecture, Wolfgang Mauerer

The abstraction for timer provides the ability to handle the differences in timer implementation between kernel and user-space. The user-space makes use of POSIX timer APIs for arming, disarming and fetching the state of per-process timer, as opposed to kernel timers, defined under Linux/timer.h for declaring, registering and removing kernel timer instances.

Another important layer of abstraction is the Inter Process Communication, which is based on socket interactions. This abstraction is performance critical and plays an important role in abstracting the message queuing subsystem and for transferring data between client and server.

As simple as it may sound, each of the abstraction layers have been designed to work with a simple set of data structures with their corresponding function calls, that hide the semantics of the underlying operating system. For example, the socket layer is abstracted using a data structure that hides the underlying differences in the socket semantics between the kernel and user-space, from the higher level components. These components invoke the new APIs designed around the socket object.

4.4 Features and Functionalities

Some of the features and functionalities in the in-kernel file sharing server, implemented in this thesis, are:

1. Ability to query the server for a list of all available resources, to locally mount on the client.
2. Capability to authenticate a remote user, by either mapping to an existing user in the system, or by providing access to the share resources as a "guest" user, with or without password, based on the server configuration.
3. Basic file system functionalities such as reading, writing, deleting and renaming files or directories, creating symbolic links, and many other functionalities a generic file-system would provide, for clients accessing a disk share.
4. Securing communication between the client and server by signing and encryption.
5. Capability to dynamically add, update and remove disk shares on the server, without the need to restart the server.
6. Capability to preserve the status of a file operation using resilient handles, when the client suffers a short network outage.
7. Cross platform interoperability with clients such as Mac OS, CIFS (Linux), Windows, and any other client as long as it adheres to the SMB specifications.

8. Ability to run as a privileged, as well as a non-privileged process based on the server format. For example, kernel-space servers always run with root privileges, while user-space servers may or may not.
9. Ability to leverage client-side caching capabilities using partial oplock support to avoid unnecessary network bandwidth consumption.

5 Benchmarking and Performance analysis

The objective of this chapter is to evaluate the newly implemented file sharing server. We study different methodologies that have been employed, in order to evaluate both the functional and the performance aspects of the server. In our case, software testing primarily refers to evaluating the application to understand its capabilities, and to determine whether the implemented in-kernel server meets the expected requirements and results.

The chapter is organized as follows. Section 5.1, introduces the basic test methodologies used during and after software development, to ensure quality of the developed product. Section 5.2 briefly defines the test environment and provides information on the hardware, software, network conditions and storage media used for benchmarking. Section 5.3, provides a detailed description on why certain decisions were made when choosing the hardware and software for the client and server. It also provides a brief introduction to the automated benchmarking suite. Sections 5.4 and 5.5, provide a description on the types of benchmarking done in this thesis, and list the metrics and the result of the respective experiments.

5.1 Test methodology

There are a large number of test methodologies that serve different purposes, depending on the phase of software development. Based on the purpose of testing, we broadly classify the evaluation methodologies into *Functional & correctness testing* and *Performance testing*.

Functional and correctness testing

It is necessary to ensure that the implemented SMB file sharing server is at least able to fulfil its minimum requirements. Through these functional tests, we can ensure that the server behaves in an expected way, and does not digress from its actual purpose. To ensure that different scopes of software testing such as unit testing, integration testing, system testing and functional testing are covered, black-box testing methodology is employed. A black box testing methodology is one where the inner workings of the application are not taken into account, and the emphasis is on executing routines that evaluate functional requirements. It is a data driven approach, where only the input and the output data are given prime importance, and the implementation details are not considered. One of the biggest advantages, and at the same time a limitation, of the black-box testing methodology, is that it presents an exhaustive list of data input and output possibilities. Thus, it is all the more easier to get lost in different permutations of the two.

For the purpose of continuous integration testing, we have developed a central testing framework that is directly synced with the version control system. This central testing framework continuously monitors new commits, bundles the commits into build artefacts on a daily basis, and triggers a series of automated test-tools meant to test the functional aspects of the SMB server. Once the tests are complete, a notification is sent to the developers, in the SMB server team, of the available summary of results. Any deviation from the expected tests are highlighted as important. Some of the testing tools used for functional testing are xfstest[26], smbtorture[27] and microsoft test suite[28]. The xfstest suite and smbtorture test framework have been modified heavily, and about 200 new test groups and more than 1000 individual test cases have been added in addition to the existing ones. The tools are being continually improved upon to cover more test cases. Some other tools are also used for negative protocol testing, to analyse security risks in the implementation and design of the SMB server.

Performance testing

Key importance must be given to the design and quality of the software architecture during implementation, since they form the major variables in an equation to satisfy key software attributes such as security, maintenance, reliability and performance. The implicit requirement for any performance testing is that the server application, while conforming to all the functional requirements, should be able to take less time

[26] https://wiki.samba.org/index.php/Xfstesting-cifs
[27] https://wiki.samba.org/index.php/Writing_Torture_Tests
[28] https://github.com/Microsoft/WindowsProtocolTestSuites

while consuming less resources during the execution of tasks. Performance evaluation helps identify bottlenecks, while covering various aspects such as throughput, request latency, CPU cycles consumed, rate of disk access operations, memory usage, etc.

The following sections will provide a performance evaluation between user-space Samba server (a widely used open source solution) and the newly implemented in-kernel SMB server. It will briefly describe the performance testing frameworks, along with workload characteristics that are designed to be a representative of how the system performs under various conditions.

5.2 Experimental Methodology

This section gives a detailed description on the test environment, as well as all the components involved, for performing and collecting benchmark statistics.

5.2.1 Test Environment

Figure 24 shows an overview of the testbed configuration used for investigating the performance of the kernel mode TSMB server.

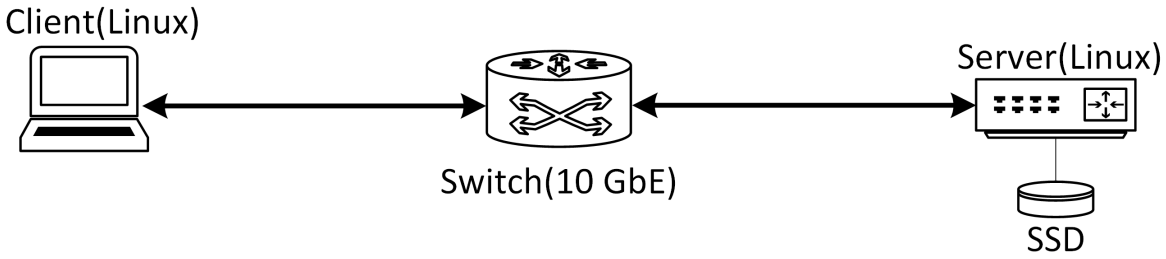


Figure 24: Testbed configuration for performance evaluation

Hardware

The testbed consists of three main components. These are, the client test machine, the server machine and finally, the switch that connects the two entities.

The client test machine is a Lenovo X250 laptop with an Intel Core i7-5600U running at a clock speed of 2.6 GHz. It has 8 GB RAM and more slots for additional memory. It is equipped with an on-board Intel I218-LM GbE controller. In order to avoid any interference with the performance testing over this interface, an additional USB 3.0 to Ethernet adapter was attached to provide remote shell login access for the purpose of debugging and collecting performance statistics. To avoid bottlenecks due

to limitations on the storage disk's read and write performance, Samsung's client edition Solid State Drive (SSD), with a capacity of 512 GB, was connected internally via a serial ATA interface.

The Device Under Test (DUT), is a router that features a Qualcomm Atheros AP148 reference board. The reference board includes a high performance dual-core ARM based CPU, namely IPQ8064 @1.4GHz and 500 MB of RAM. It has 5 high performance GbE ports, one of which is connected to the test environment. The device also has 2 USB 3.0 interfaces. On one of the USB 3.0 interfaces, a random storage drive is connected to house all the important server softwares and utilities, required for invoking a server instance. The other USB 3.0 has a high speed Samsung 850 pro 128 GB SSD. The device also has a serial converter which can be configured to provide a serial console for debugging and accessing the device. One of the notable features about this open reference board, is that it provides two multi-threaded network accelerator cores @733MHz which allows for network input/output acceleration and many offloading capabilities. The IPQ8064 is a popular system on a chip from Qualcomm Atheros, and is found in many high-end consumer routers.

Both the client laptop and the DUT support a 1 GbE controller. The two machines are connected by a HP 2920 48G switch with throughput upto 130.9 Million packets per second (Mpps), switching capacity of 176 Gbps and four optional 10 Gigabit ports (SFP+ and/or 10GBASE-T).

Software

The Linux client laptop runs an Ubuntu Xenial 16.04.1 LTS operating system based on the Linux kernel[29] 4.4.0. Linux version 4.4.0, by default supports an SMB client, namely CIFS. The CIFS is a client-side implementation of the SMB protocol, and is actively being developed as part of Linux kernel[30]. The latest version of the CIFS file system 2.08, is used for mounting the remote shares exported by the server. This mounted share point in the client is used for benchmarking the performance of the server under certain workloads. The Solid State drive connected to the client machine is formatted with the Ext4[31] file system. Even though a large number of file systems are available for flash-based storage media on Linux, we use Ext4 since it is considered the de-facto file system on Linux.

For the DUT, the latest available firmware from the vendor for the reference board has been chosen. The readily available firmware is a custom compiled Linux OpenWRT version 3.4.103. The SSD connected to the router is once again mounted with an

[29] https://www.kernel.org/
[30] https://wiki.samba.org/index.php/LinuxCIFSKernel
[31] https://ext4.wiki.kernel.org/index.php/Main_Page

Ext4 file system. The directory on which the storage media is mounted is used for running a performance benchmark on the server. One of the drawbacks of using the reference boards, is that the operating system and the software compiled to run on the device, are severely limited by the available tool-chain[32].

In the DUT, two software servers implementing the SMB protocol are tested, namely Samba and the in-house kernel-mode server. Due to the limitations imposed by the available tool-chain for the build environment, Samba version 3.6.25 was compiled. During the time of writing the thesis, the highest stable version of samba was 4.4.5, but compilation of any higher version has issues, such as lack of python libraries in the given tool-chain. For the in-house kernel mode server, we limit the version of the server to a proof-of-concept model that was completed during early March 2016. Newer versions with performance and feature enhancements are not evaluated in this thesis, due to a strict non-disclosure agreement.

Network conditions

We have chosen a 1 GbE network to evaluate the performance of the software servers. The network conditions are kept ideal and no external emulators are used for modifying the network latency, jitters or packet loss, as evaluating the performance impact of varying network conditions on the servers, is beyond the scope of this thesis. Due to popularity of the server hardware in consumer routers, we are assuming that the performance evaluation will be close to the access links available to home users.

Storage target

In this test-bed configuration, there is only one storage target. A storage target can be defined as a mount point, which is a directory or a folder currently residing on the client's local file system, on which an additional file system is mounted. The storage target that will be used for performance evaluation is a directory mounted with the SMB protocol based network file system.

On the DUT, the exported software target is generally a directory on the local file system, stored on the SSD attached to it.

[32]The tool-chain is a software package needed to compile, link and deploy software applications from the host used for development to the target device.

5.3 Performance Benchmarking

Benchmarking is an essential process that measures and compares performance against a set standard. Normally, the interactions between IO devices, kernel daemons, threads, and other OS subcomponents, make it very difficult to expect a fixed behavioural aspect during benchmarking. Therefore, due to such complexities in the nature of network storage systems, no single benchmark can gauge every aspect of a server.

There are an overwhelming number of benchmarking tools available, many of which either over emphasise the nature of some overheads, or conceal them[30]. Due to the huge amount of benchmarking data, one also tends to easily get lost in details. Therefore, it is very important to define a scope for benchmarking. By defining a scope, we clearly define the following aspects. The specifics of the server application that will be evaluated during benchmarking, the reason for ignoring certain aspects while benchmarking, the behaviour of the server under different synthetic and realistic workloads, and most importantly provide the reader the ability to verify the benchmark results.

5.3.1 Choice of client

As discussed earlier, the test bed configuration uses a Linux client for benchmarking the server. There are multiple clients available which provide support for SMB protocol, with the most popular being Windows followed by Linux. Though Windows is the most popular choice, the variety of tools available for creating a refined benchmark tests are limited. Familiarity with Linux and a plethora of available benchmarking tools[30], make it an obvious choice for performance evaluation.

5.3.2 Choice of server

The primary motivation behind choosing an embedded open reference board over a custom built high-end server-grade desktop, is that even in an environment that limits the available resources, the in-kernel server should be able to provide a higher performance compared to its user-space alternative.

Secondly, during our initial investigation, we determined that the overhead introduced due to system calls, boundary crossing or extra memory copies across kernel and user-space, and the extra CPU utilization because of context switching, does not matter when the server is not running at the edge of the hardware limit. Due to the availability of extra CPU time on high end servers, the added overhead of moving to user-space does not show any significant performance difference between the kernel

and user mode servers. This was visible during our testing phase, but we haven't conducted any extensive study to prove this.

Keeping the above reasons in mind, we selected an embedded platform which is popular in general home networks, and where file sharing servers are being deployed more aggressively.

5.3.3 Scope of benchmark

Network storage systems have several aspects that can be benchmarked. Hence, the number of tests for benchmarking must be extensive enough, to cover different subsystems of a software server implementation. Due to the extremely wide nature of available tests, we will be limiting the scope of benchmarking as discussed below.

In order to provide a meaningful server performance benchmarking, we will compare the in-kernel server to an existing implementation, namely Samba. Since both Samba and the in-kernel server are CPU and IO bound applications, we attempt to focus on providing a benchmark which gives a low level view of different performance aspects of the server, and at the same time, monitoring how much CPU resource is consumed during each scenario or workload. For a low level view, we employ both micro benchmarking and meta-data benchmarking, since they help isolate specific subsystems within the server application.

As we are using an embedded platform that is widely popular in home networks, we will refrain from conducting any scalability or server resiliency benchmarking. Scalability benchmarking is meant to test how well the server is scalable with increased client connections, while server resiliency benchmarking tests whether a server is capable of handling network scenarios which introduce packet loss or latency. The performance benchmarks are mainly aimed at how well the server can handle IO requests from a client, and the amount of CPU resource expended in handling these IO requests.

5.3.4 Benchmark environment

It is very important to understand the state of the system before and during the benchmark, since it can have a significant impact on the observed results. Some of the factors that affect the state of the two systems, namely the client and the server in our test-bed configuration are, the age of the file system i.e. how frequently the local file system has been accessed by IO requests, the age of the file sharing server, the state of the cache and lastly, other unessential processes that might be running at the time of benchmarking.

As mentioned earlier in this chapter, the benchmarks are run under ideal conditions. Keeping this in mind, the local file system on the DUT is always formatted newly before a benchmark. This formatting is done in order to avoid any hindrance due to file system ageing, that can be caused by continuous IO activity on the file system. Another important aspect to keep in mind, is that the disk partitions are aligned, perhaps at 1 MB boundaries. This ensures that the file system blocks are aligned with the sectors on disk, and as a result can avoid IO amplification. The storage media is formatted with ext4, with journalling capability disabled. By disabling journalling, we avoid any extra disk access introduced as an overhead for maintaining the sanctity of the file system, in case of power failure or unexpected device resets. Since we are using a controlled environment for benchmarking, we are at liberty to safely disable this feature.

With respect to the state of cache on the system, we make sure that the caches are always cold. *Cold* caches refer to flushed or empty memory, and *warm* caches are those with some amount of data still in memory. If caches are kept warm, data access may be done from the caches, resulting in inconsistencies in the observed results. For this reason, we maintain cold caches by discarding the caches, reformatting and remounting the file system and restarting the file sharing server across each test runs. This ensures that we execute identical test runs and are able to observe stable test results.

Finally, to ensure that other unnecessary processes and services do not interfere with the benchmark runs, they are terminated. The observed CPU usage on both the client and server machines, are kept well under approximately 0.2% when the system is idle.

5.3.5 Benchmark Design

We designed a custom benchmark to simulate an application performing an IO on the disk. In order to keep the tests consistent across benchmarks, we introduced an automated benchmarking suite capable of generating sequential access patterns, based on shell script. The user running this tool has to invoke the server script on the DUT, and then the client script on the client laptop. This order of invoking this suite introduces a certain degree of automation.

We have already established that the tests are run with cold caches on a fresh server. The server script flushes and clears the local cache, formats and mounts the local file system and invokes an instance of the file sharing server to be benchmarked. The server script listens for a START command over the network, made possible using the **netcat** utility. On receiving the start command, the server waits for one second and then samples CPU usage metrics at one second intervals, until the test ends.

The end of a test run is signalled by a STOP command to the DUT, from the client script. On receiving the stop command, the server ignores the last sample of the collected metric, and generates an average CPU usage for the duration of the test run. The last metric is ignored to overcome the deviation introduced by the delay between receiving the stop command and stopping the test running on the server.

The client script takes care of flushing the local cache, and remounting the network file system on a directory on the local file system. Once the mounting is successful, the client script sends a start command to the server and waits for one second before running the tests. On the client side, metrics are collected by the script based on the test being run. The client ensures that the test runs for at least 2 seconds, ignoring all tests that do not meet this criteria. Each of the benchmark results shown in the later sections, are an average of 15 test runs.

To ensure that we do not end up benchmarking the Linux kernel caching ability, we always flush and drop the kernel cache using the following command:

$$echo \quad 3 > /proc/sys/vm/drop\_caches \tag{1}$$

Tweaks to CIFS client and Samba server

SMB file sharing protocol supports a capability known as *oplocks*. Opportunistic locks provide exclusive locking rights to the client for the files residing on the server, thereby allowing client-side caching abilities. This boosts the performance, by caching read and write requests on the client without the need to send data to the server. This feature also coalesces smaller read and write requests, such as combining 4KB chunks into larger chunks of 64 KB, and then sending the request to the server. This will severely skew the observed results in some of the tests, and therefore, this ability is disabled on the client, using a mount option *cache=none*.

Though the protocol specification provides a theoretical limit of 8 MB on the maximum request size of certain commands of SMB2 protocol, the design of the in-kernel server further limits the maximum request size to 64 KB. We limited the maximum request size supported by the open source Samba server to 64 KB, using the following options in the Samba configuration file:

$$smb2 \quad max \quad trans = 65536 \tag{2}$$

$$smb2 \quad max \quad read = 65536 \tag{3}$$

$$smb2 \quad max \quad write = 65536 \tag{4}$$

We also enable *use sendfile = yes*, configuration in Samba. This optimizes Samba's read performance by avoiding unnecessary data copy from kernel to user-space, using *sendfile()*. Using this API, it is possible to directly send data from a file onto the network socket. We believe that these configurations have the most impact on Samba's performance.

The above steps were taken in order to have a fair benchmark testing between Samba and the in-kernel server.

5.4 Micro Benchmark

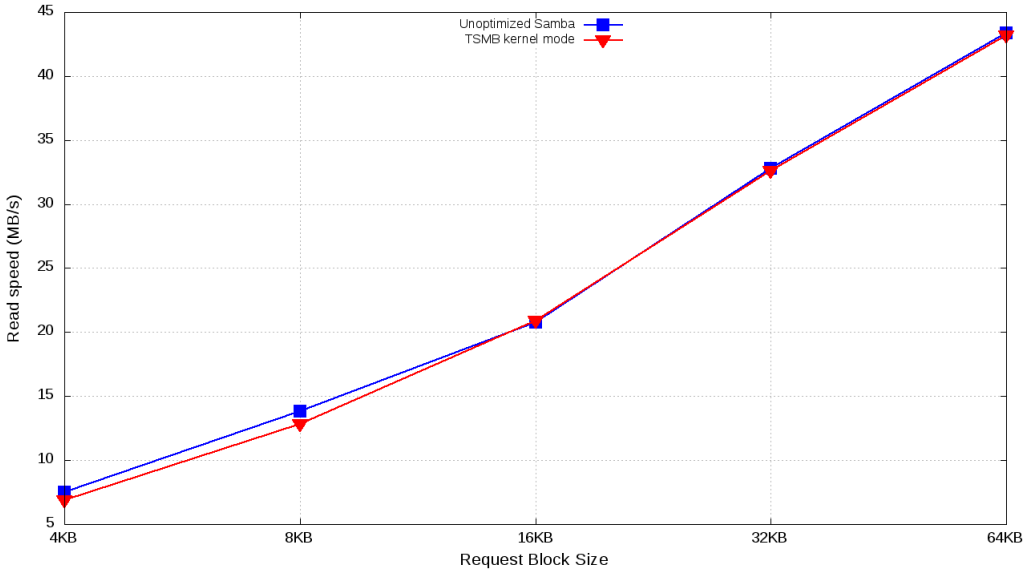
In this section, we discuss the impact of using different block sizes for accessing data over the network file system, and compare the performance of Samba against the in-kernel file sharing server. As mentioned earlier, in order to ensure that smaller requests sizes are not coalesced, we disable the oplock feature supported by the protocol. The experiment is conducted by accessing a file which is 2 GiB in size, using access blocks ranging from 4 KB to 64 KB.

The metric collected by the client script for this benchmark, is the speed of read and write operations in MB/s. This is directly co-related to the time taken to write a 2 GiB file using a selected block access size for a single benchmark run. On the server side, we collect the CPU usage from the relevant Linux counters using *mpstat and /proc pseudo file system*.

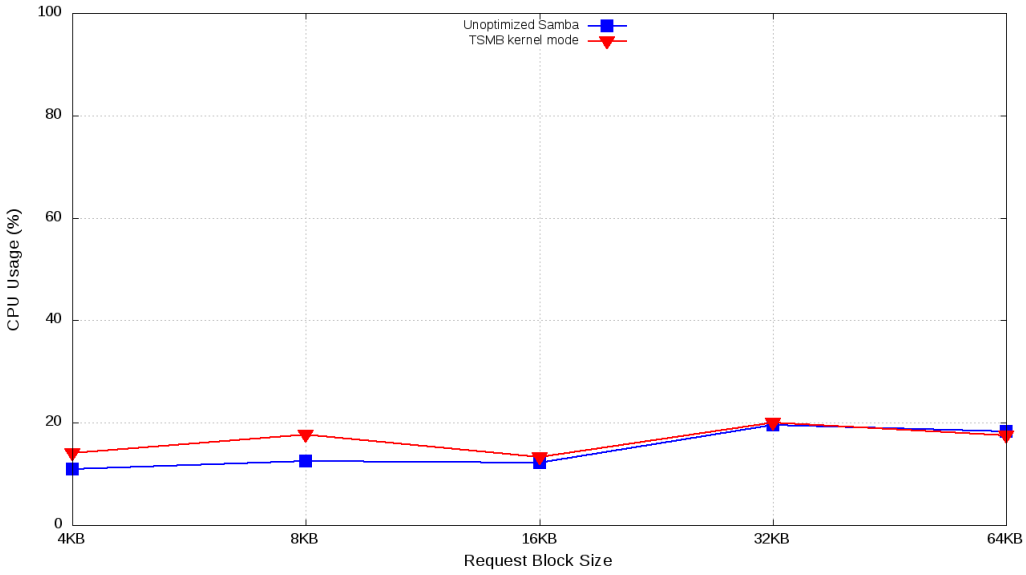
As discussed earlier, the caches on the client and server are always flushed before each test run. For conducting the write test, we copy a 2 GiB file from the client to the server. For the read test, we read the same file from the server to the client. The read file is redirected to */dev/null* device, so as not to incur additional overhead in creating a local copy of the file in the client.

A comparison of read performance for different access block sizes are illustrated in Figure 25. As mentioned earlier, the use of sendfile configuration in Samba enhances its read performance. This is especially clear with Samba's read speed being slightly higher than kernel mode server, at least for smaller block sizes. Here we can say that the user-space Samba server is clearly performing better compared to the in-kernel server, but as the block size increases, the performance is almost similar. The most significant difference in read performance is visible with the CPU utilization. The kernel mode server clearly uses more CPU compared to the user space server. With lower block sizes, the kernel mode server is taxed more due to multiple buffer copies, and as a result the CPU usage significantly increases. As we reach higher block sizes, the gap between performance and CPU usage becomes smaller.

The write performance comparison for different block sizes is illustrated in Figure



(a) Read Speed

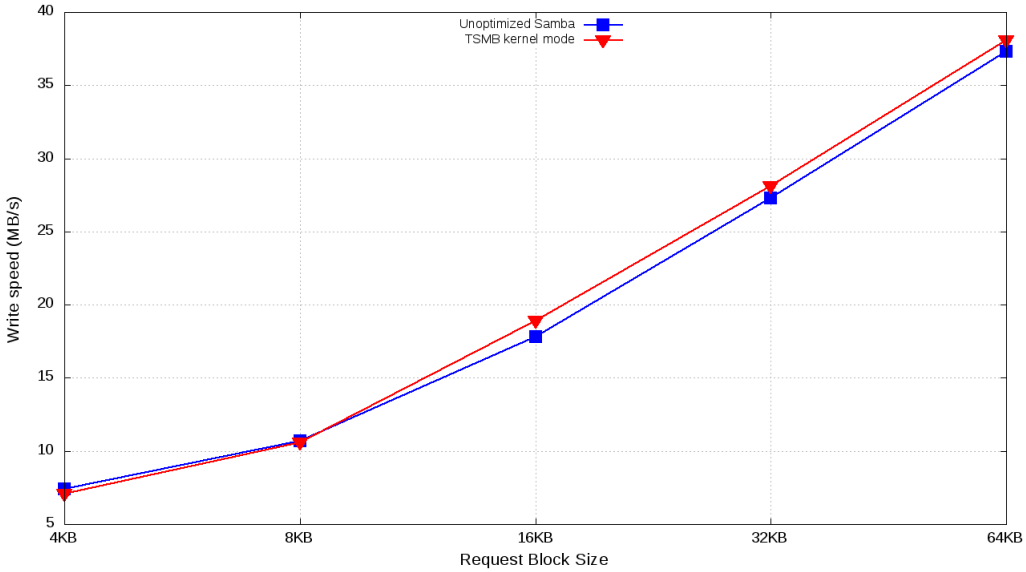


(b) Read CPU usage

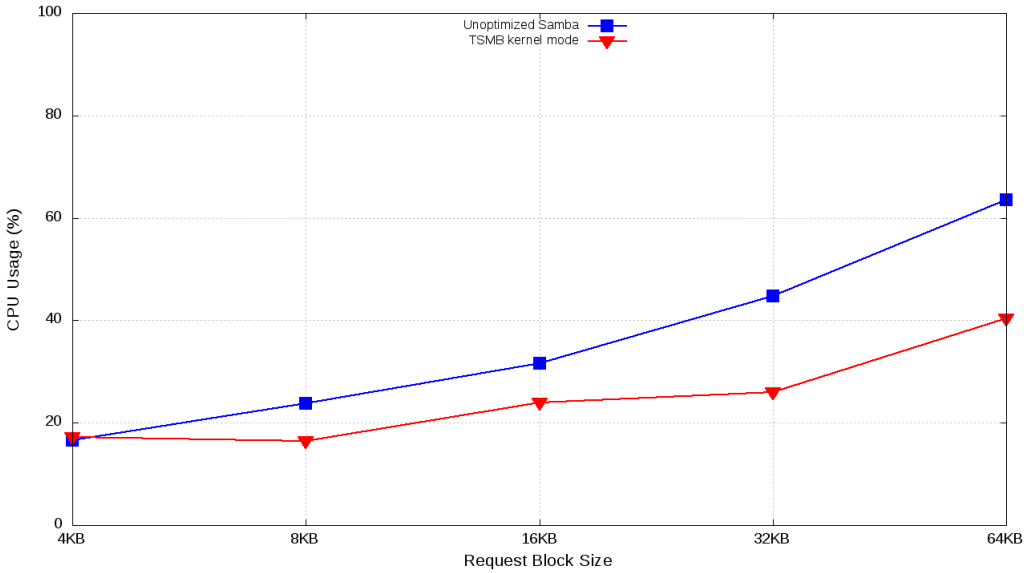
Figure 25: Read performance comparison for different access block size with test file size of 2 GiB.

26. The performance of in-kernel server is definitely better than Samba, though only slightly. It should be noted that, unlike sendfile performance enhancement for read in Samba, write does not have any such enhancement. Both the in-kernel server and Samba, copy data from socket and then invoke the corresponding function calls to write data to a file. We clearly see that the CPU usage for Samba is 60% higher than the CPU usage by the in-kernel server. With increased CPU usage and lower write performance, the user-mode server is definitely at a disadvantage compared to

the kernel mode server.



(a) Write Speed



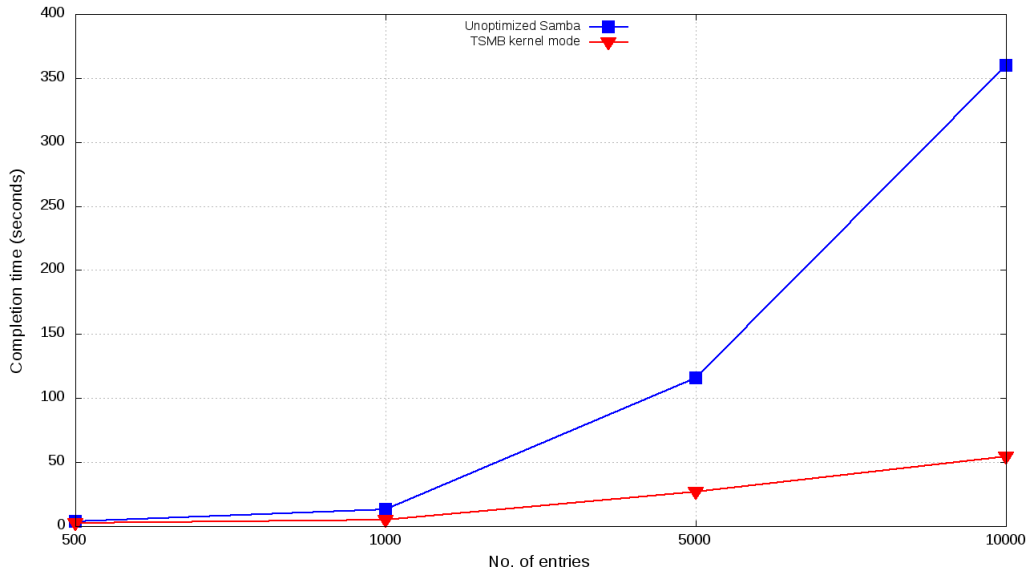
(b) Write CPU usage

Figure 26: Write performance comparison for different access block size with test file size of 2 GiB.

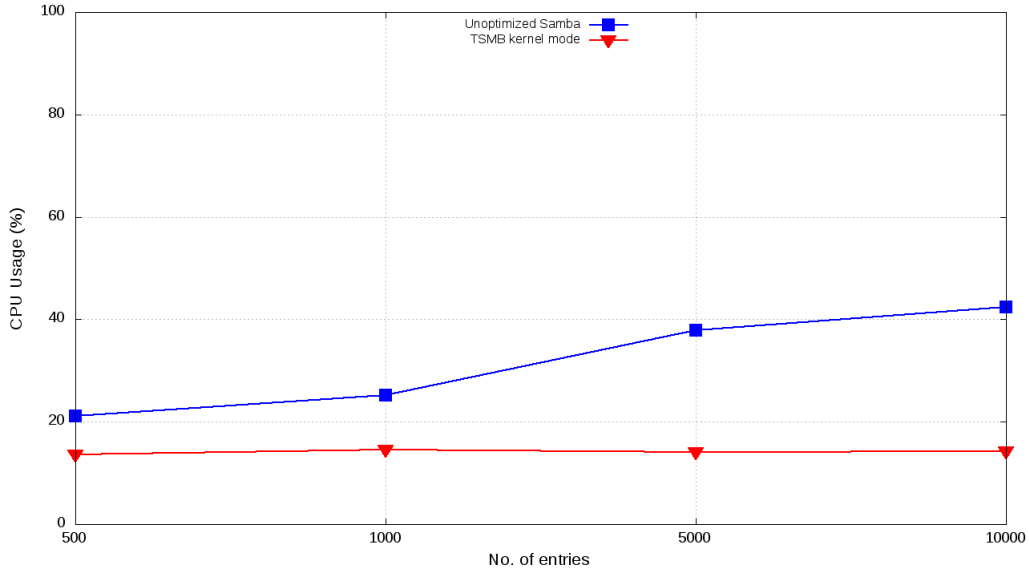
We believe that if the sendfile configuration in Samba were disabled, its read performance would be much lower than the in-kernel server, with the the CPU usage pattern being very similar to the one observed for write operation, as shown in Figure 26.

5.5 Metadata Benchmark

In this section, we compare the performance between Samba and in-kernel server based on metadata operations. For this purpose, we use the same automation script as defined in Section 5.3.5, except for the change in benchmark routines.



(a) Create Speed

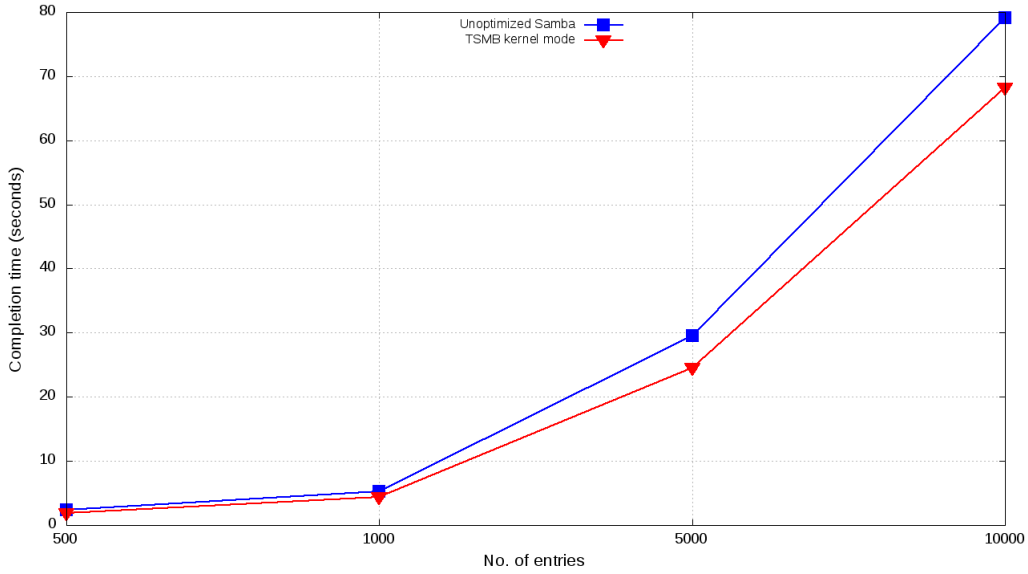


(b) Create CPU usage

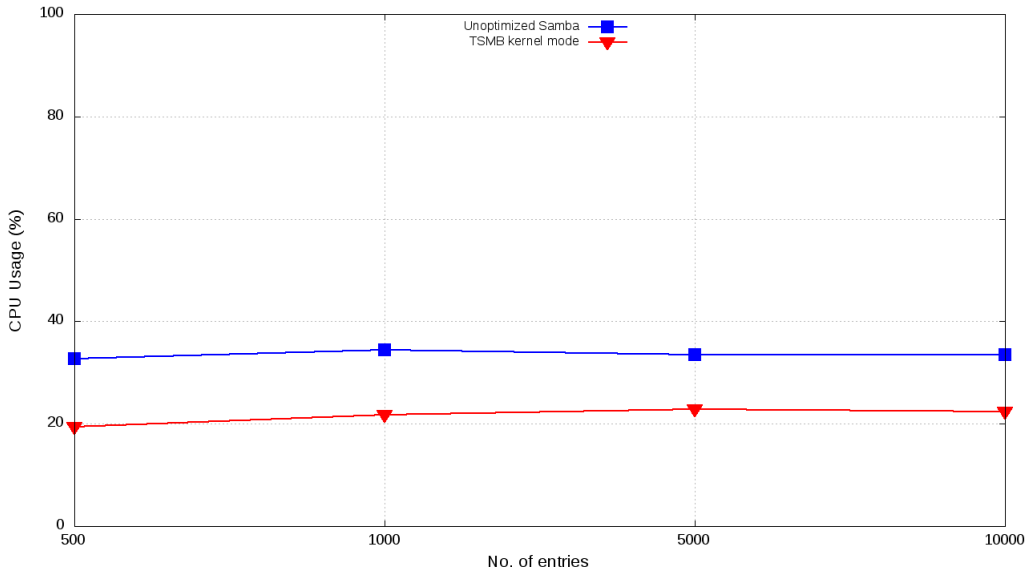
Figure 27: Completion time for create(mkdir and touch) operations

For this benchmarking, the client script invokes a metadata routine, where we execute operations such as creating, listing and deleting. In create test, *'n'* entries are generated, where *'n/2'* entries are empty directories created using **mkdir** command,

and *'n/2'* entries are empty files created using **touch** command. Entries are listed using the **'ls -lah'** command and deleting the entries are done using **rmdir** and **unlink** respectively. The metric collected by the client script is the completion time in seconds, for each set of metadata operations. The server script, for the metadata benchmark routine, collects the CPU usage statistics till the operation ends. The end of operation is signalled from the client to the server as discussed earlier.



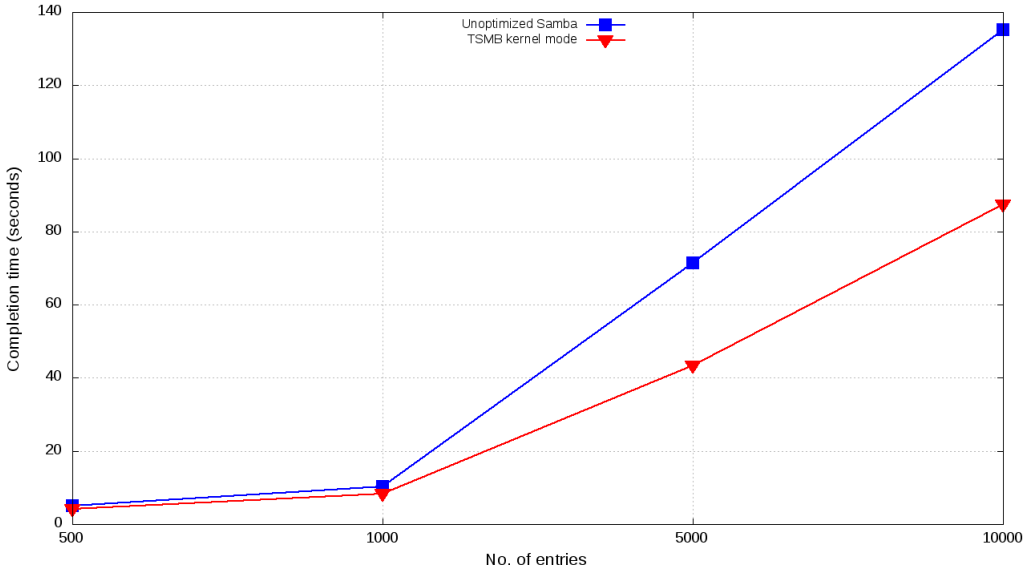
(a) Listing Speed



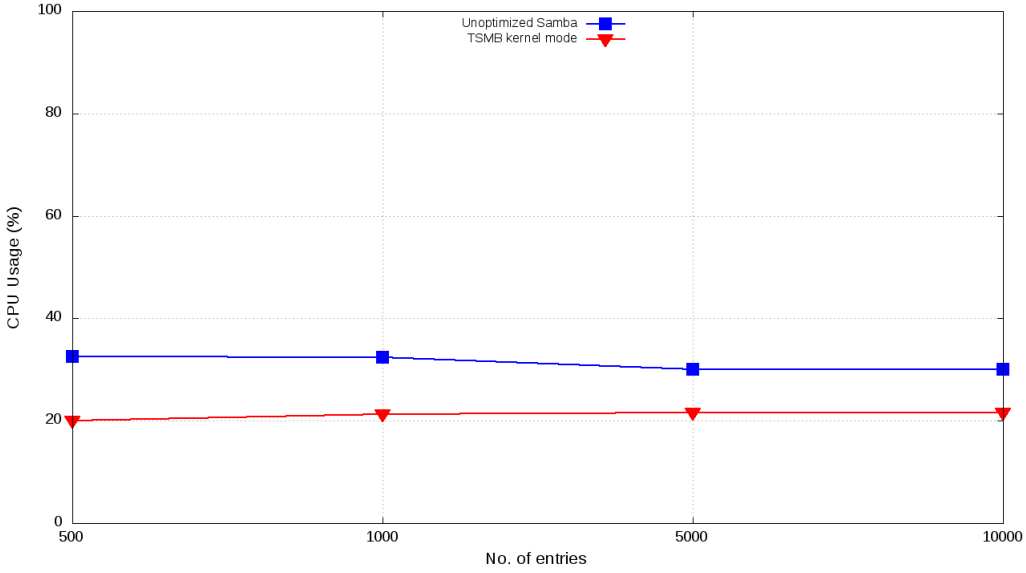
(b) Listing CPU usage

Figure 28: Completion time for listing(ls -lah) operation

The comparison of the performance based on completion time and CPU usage for create operations is illustrated in Figure 27. We can see from the illustrated graphs



(a) Delete Speed



(b) Delete CPU usage

Figure 29: Completion time for delete(rmdir and unlink) operations

that the kernel mode is about 7 times faster compared to Samba, when creating directories and files. It is also visible that the CPU usage is only around 15% for in-kernel server, whereas Samba takes as high as 40% CPU while performing the same task for a longer period of time.

The Figure 28, illustrates the performance based on completion time for listing entries within a directory. Listing the child entries in a directory can be thought of as reading the contents of the directories, which is IO intensive. We can clearly see

that once again the in-kernel server is able to list entries faster than its user-space counter part, while consuming 54% less CPU.

The performance comparison based on completion time of removing directories and files, is shown in Figure 29. The interesting observation here is that like other metadata operations, the in-kernel server is able to handle the delete operations more efficiently, when compared to Samba. Here we can see that the in-kernel server takes around 58% percent less time to delete entries, while expending very low CPU resource compared to Samba.

6 Summary

This chapter reviews the tasks undertaken during the course of this thesis, and summarizes the findings and conclusions. Section 6.1 provides a conclusion of the project and summarizes the results and findings, and Section 6.2 provides an overview of all possible activities planned to improve the performance of the in-kernel file sharing server, in the near future.

6.1 Conclusion

We successfully provided an implementation within 6 months of starting the development activities, with all the basic support for SMB1 and SMB2 protocol versions. The design of the server architecture has been planned in such a way that the kernel mode server can also be compiled to function in user-space, if the need arises. The main motivation behind this thesis, is to determine whether we can improve performance when moving the server implementation to kernel space. Generally speaking, under ideal network conditions, the kernel mode server definitely outperforms Samba, while using significantly less CPU cycles in most of the tested cases.

Samba and the in-kernel server show the greatest difference in their handling of meta-data operations. From the results shown in Section 5.5, we observe that the kernel mode server is able to perform better than Samba for meta-data intensive workloads. So if a user were to create thousands of files or directories, or attempt to list all the available files within a directory, the in-kernel server would be able to return data much faster compared to Samba. Contents of the directories can be listed 17% faster by the in-kernel server compared to Samba, as shown in Figure 28. This performance is significant especially if a user wants to browse a directory with millions of documents, and wants to read one particular document. The in-kernel

server is able to handle such meta-data operations faster, without being overwhelmed by processing meta-data requests.

With data intensive payload, we find two different observations for read and write speeds. Figure 25 shows that Samba is able to get a slight edge over the in-kernel server for read speeds with block sizes less than 16KB. This observation can be associated with the use of sendfile API by Samba. The sendfile API copies data from one socket descriptor to another file descriptor in the kernel-space. Since this copy is done in the kernel-space, it is more efficient than issuing *read()* to read data from file, and then using *write()* system call to write data to socket. Using sendfile not only reduces the system calls by half, but also avoids unnecessary data copy between kernel and user-space. Despite the advantages of using sendfile, Figure 25 shows that both kernel and user-space have very similar performance in terms of both read speed and CPU usage, as the block size increases. In the in-kernel server, we read all the data from a file to an internal buffer, which is then passed across different components before it is sent back to the client. We believe, that by avoiding the unnecessary buffer copying between the different components within the kernel mode architecture, we will be able to boost performance significantly. With respect to data intensive write workload, since both Samba and the in-kernel server follow a very similar read/write pattern, we notice that the in-kernel server is able to outperform Samba, both in terms of speed and CPU usage.

We clearly see, that moving to kernel space has given a slight edge, and sometimes a very significant boost in performance over its user-space counterpart. It is important to note that the version of kernel mode server used in this thesis, was the initial proof-of-concept implementation, which does not have many performance improvements identified during the initial phase of development. It is also important to keep in mind, that kernel space is supposed to be kept minimal, i.e. the design should be done in a way that only CPU and IO intensive tasks, which can truly benefit from integrating tightly with kernel space, are the only components to be moved to kernel space, while keeping the large chunk of support services in the user-space. By making use of a split architecture, we believe that even with this minimalistic version that was evaluated in this thesis, we were able to show that moving file sharing servers to kernel space is definitely the right direction to go, while continually identifying and solving newer challenges.

6.2 Further developments

Although we seem to have achieved a fully interactive and functional network file system with the in-kernel server, many of the niche features are still unimplemented. With the SMB protocol being continually improved, it presents unlimited room for

further developing and improving the features and performance of in-kernel SMB server.

Zero Copy

The kernel module serves as an intermediary, where the data is copied from a socket descriptor, maintained in an internal buffer and then copied to a file descriptor, and vice versa. Each time data traverses through this cycle, data copies are made before sending the data to the appropriate destination. These extra data copies can be eliminated by using the sendfile approach, where the data from a file is copied onto a kernel buffers using DMA copy. The kernel buffer is then copied onto socket buffers, after which a DMA copy is made from the socket buffer onto the network card buffer, which indicates the end of transfer. This sendfile approach can be further optimized by using just two DMA copies, if the hardware is capable of doing a DMA scatter/gather where it is able to map the list of buffers at once and then transfer the list of mapped pages in a single DMA operation[33]. With this approach, the in-kernel server should be able to see a significant boost in performance.

Large block access size

SMB 2.1 protocol dialect, sports a feature for significantly improving the performance of the protocol on high speed and low latency networks, by introducing a feature called large MTU or multi-crediting. With this feature the maximum transmission unit, which is representative of the largest protocol data unit that the SMB protocol can communicate across with the remote entity, was increased from 64 KB to theoretical maximum of 8 MB, even though only 1 MB large PDUs are supported by Windows client. The latest version of the in-kernel server supports this.

Resource management

Memory management plays a significant role when it comes to the performance of a file sharing server. Due to the constant flow of packets, a server is continually allocating and deallocating memory buffers ranging from few bytes to couple of mega bytes of memory for parsing, copying data to file, and a myriad of other operations. When servers run for a long period of time, be it in kernel or user-space, they tend to cause extreme memory fragmentation. With fragmented memory, it is possible that when other applications attempt to allocate memory, the operation can fail leading

[33]Chapter 15: Memory Mapping and DMA, Linux Device Drivers, O'REILLY, Feb 2005, page 450.

to erratic behaviour by the system services. This was observed when functional and performance testing was run continuously for more than two days on both Samba and the in-kernel server. By implementing resource management features such as memory pools, it is possible to control fragmentation caused by long running IO intensive network applications.

Concurrent operations

Though the current implementation of kernel mode server is multi threaded, with each thread handling its own respective tasks, there are still ways to improve the concurrency of operations. For example, by increasing the number of protocol engine and transport threads, each new client can be handled by an individual thread. This is especially useful if there are multiple processors/cores. Then each thread can reside on a different core. Increasing the number of TSMB VFS threads can also increase the simultaneous IO access to disks. We believe that by implementing component specific thread pool, we will be able to spread the load and provide highly concurrent operations.

Expanding SMB feature set

The SMB2 protocol specification is continually being worked upon, with each new feature addition bringing with it, a vast change in the underlying technology covering various aspects of functionalities such as clustering, transparent server fail-over in a clustered environment, performance enhancements that increase the performance of a file server through technologies such as SMB Direct[34], and other performance enhancing features such as Multichannel, which does link aggregation to increase throughput using multiple connections. Apart from the above mentioned features, SMB protocol provides a plethora of other capabilities which require a lot of time and effort, if one were to support the complete feature set.

[34]SMB Direct uses Remote Direct Memory Access (RDMA) capable network adapters to use SMB over RDMA instead of SMB over TCP/IP

References

[1] Gantz J F, Reinsel D.,*THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East*, An Internet Data Center (IDC) White Paper, sponsored by EMC, Dec 2012.

[2] Andrew S. Tanenbaum, *Modern Operation Systems*, Chapter 4 File Systems, pages 263-336, Edition 4, Pearson Education Limited, 2015.

[3] Batchanaboyina et al., *Recent Trends on DAS and SAN Protocols*, Namoori P. B., Devarapalli A. and Ramaiah T., *International Journal of Advanced Research in Computer Science and Software Engg*, 5(4), pages 563-567, April 2015.

[4] Qunhui Wu, *Research on the application of storage technology in Digital Library*, International conference on *Future Information Technology and Management Engineering (FITME)*, Vol 2, pages 166-169, 2010.

[5] C. DeCusatis, *Storage area network applications*, *Optical Fiber Communication Conference and Exhibit, 2002. OFC 2002*, pages 443-444, 2002.

[6] Steven M. French, Samba team, *A New Network File System is Born: Comparison of SMB2, CIFS, and NFS*, Proceedings of Ottawa Linux Symposium, pages 131-140, Available: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-131-140.pdf, 2007.

[7] J. Norton, et al. *SNIA CIFS Technical Reference*, Available; http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf, March 2002.

[8] *[MS-CIFS]: Common Internet File System (CIFS) Protocol*, Microsoft Corportion, MS-CIFS, Rev. v20160714, July 14 2016.

[9] *[MS-SMB]: Server Message Block (SMB) Protocol*, Microsoft Corporation, MS-SMB, Miscrosoft corporation, Rev. v20160714, July 14 2016.

[10] *[MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3*, Microsoft Corporation, MS-SMB2, Rev. v20160714, July 14 2016.

[11] *[MS-SRVS]: Server Service Remote Protocol*, Microsoft Corporation, MS-SRVS, Rev. v20160714, July 14 2016.

[12] *[MS-SMBD]: SMB2 Remote Direct Memory Access (RDMA) Transport Protocol*, Microsoft Corporation, MS-RDMA, Rev. v20160714, July 14 2016.

[13] *Supplement to InfiniBand™ Architecture Specification Volume 1 Release 1.2.1*, InfiniBand™ Trade Association, Annex A16: RoCE, April 6 2010.

[14] *Supplement to InfiniBand™ Architecture Specification Volume 1 Release 1.2.1*, InfiniBand™ Trade Association, Annex A17: RoCEv2, September 2 2014.

[15] *InfiniBandTM Architecture Specification Volume 2, Release 1.3*, InfiniBand^{SM} Trade Association, Nov 6 2012.

[16] R. Recio et al. *A Remote Direct Memory Access Protocol Specification*, RFC 5040, http://www.rfc-editor.org/rfc/rfc5040.txt, October 2007.

[17] H. Shah et al.(October 2007) *Direct Data Placement over Reliable Transports*, RFC 5041, https://tools.ietf.org/html/rfc5041, Retrieved May 4 2011.

[18] *[MS-SPNG]: Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) Extension*, Microsoft corporation, [MS-SPNG], Rev. v20160714, July 14 2016.

[19] *[MS-KILE]: Kerberos Protocol Extensions*, Microsoft corporation, MS-KILE, Rev. v20160714, July 14 2016.

[20] *[MS-NLMP]: NT LAN Manager (NTLM) Authentication Protocol*, Microsoft Corporation, MS-NLMP, Rev. v20160714, July 14 2016.

[21] *[MS-DTYP]: Windows Data Types*, Microsoft Corporation, MS-DTYP, Rev. v20160714, July 14 2016.

[22] *[MS-RPCE]: Remote Procedure Call Protocol Extensions*, Microsoft Corporation, MS-RPCE, Rev. v20160714, July 14 2016.

[23] *[MS-FSA]: File System Algorithms*, Microsoft Corporation, MS-FSA, Rev. v20160714, July 14 2016.

[24] Philippe Joubert, Robert B. King, RichNeves, Mark Russinovich, John M. Tracey*High-Performance Memory-Based Web Servers: Kernel and User-Space Performance*, Proceedings of the General Track: USENIX Annual Technical Conference, Pages 175-187, 2001.

[25] Amol Shukla, Lily Li, Anand Subramanian, Paul A.S Ward, Tim Brecht, *Evaluating the performance of user-space and kernel-space web servers*, CASCON '04 Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, Pages 189-201, 4 October 2004.

[26] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. *Comparing and evaluating epoll, select, and poll event mechanisms*, In Proceedings of 6th Annual Linux Symposium, July 2004.

[27] Chuanpeng Li, Chen Ding, Kai Shen, *Quantifying the cost of context switch*, Proceedings of the 2007 workshop on Experimental computer science, p.2-es, June 13-14, 2007.

[28] Francis M. David , Jeffrey C. Carlyle , Roy H. Campbell, *Context switch overheads for Linux on ARM platforms*, Proceedings of the 2007 workshop on Experimental computer science, p.3-es, June 13-14, 2007

[29] Tomas Hruby, Teodor Crivat, Herbert Bos, Andrew S. Tanenbaum, *On Sockets and System Calls Minimizing Context Switches for the Socket API*, TRIOS'14 Proceedings of the 2014 International Conference on Timely Results in Operating Systems, Pages 8-8, 2014.

[30] Avishay Traeger, Erez Zadok, Nikolai Joukov, Charles P. Wright, *A nine year study of file system and storage benchmarking*, ACM Transactions on Storage (TOS), v.4 n.2, p.1-56, May 2008.

[31] J.Corbet, A.Rubini, and G.Kroah-Hartman. "Memory Mapping and DMA" in Linux Device Drivers, third edition, O'REILLY, Feb 2005, pp.450.